kaaengine

Mariusz Okulanis, Marcin Labuz, Pawel Roman

Jul 04, 2022

CONTENTS:

Welcome to kaa - the 2D games engine for humans, written in python! You will be able to write 2D games quickly and efficiently, working with simple and intuitive interfaces that follow the zen of python philosophy. Windows is a first class citizen.

Installation:

```
pip install kaaengine
```

CHAPTER

ONE

TUTORIAL

1.1 Part 1: Engine and window

By the end of this tutorial you will code a complete game: a top-down shooter with animations, physics, sounds, basic AI, HUD display and multiple scenes. You will be surprised how easy and intuitive it is with the kaa engine.

With just about 400 lines of python code you'll build this game:

Parts 1 and 2 of the tutorial are explaining basic concepts of the engine - you shouldn't skip them, even if you're an experienced developer. The actual game development starts in Part 3.

We encourage you to make experiments on your own during the tutorial. If you get lost in the process, just check out the tutorial code - it's available in this git repository

Have fun!

1.1.1 Installing kaaengine

To install kaaengine:

```
pip install kaaengine
```

NOTE Kaaengine requires python 3.X. The tutorial assumes you're using python 3.6.X or newer.

1.1.2 Hello world!

To run a game you need to declare and create the first scene, initialize the engine and run the scene. Create a folder for your game and create a file named main.py inside the folder. It will be an "entry point" of your game. Put the following code inside main.py:

```
from kaa.engine import Engine, Scene
from kaa.geometry import Vector


class MyScene(Scene):

    def update(self, dt):  # this method is your game loop
        pass  # your game code will live here!

if __name__ == "__main__":
    with Engine(virtual_resolution=Vector(800, 600)) as engine:
        my_scene = MyScene()  # create the scene
        engine.run(my_scene)  # run the scene
```

Start the game by running:

```
python main.py
```

You should see a 800x600 window with a black background. Congratulations, you got the game running!

1.1.3 Understanding virtual resolution

Let's now explain what virtual resolution is and how it's different from a monitor screen resolution. When writing a game you would like to have a consistent way of referencing coordinates, independent from the screen resolution the game is running on. So for example when you draw some image on position (100, 200) you would like it to always be the same (100, 200) position on 1366x768 laptop screen, 1920x1060 full HD monitor or any other of dozens display resolutions out there.

That's where virtual resolution concept comes in. You declare a resolution for your game just once, when initializing the engine, and the engine will always use exactly this resolution. If you run the game in a window larger than declared virtual resolution, the engine will stretch the game's frame buffer (actual draw area). If you run it in a window smaller than declared virtual resolution, the engine will shrink it.

Let's test this feature by declaring window size different than the virtual resolution. Let's also tell the renderer to paint the frame buffer with a different color so we can see the results.

Add the following imports to your code:

```
from kaa.colors import Color
```

Then modify the block where the engine is initialized:

```
with Engine(virtual_resolution=Vector(800, 600)) as engine:
    # set window properties
    engine.window.size = Vector(1000, 600)
    engine.window.title = "My first kaa game!"
    # create a scene
    my_scene = MyScene()
    my_scene.clear_color = Color(0.1, 0.1, 0.1, 1)  # using RGBA with values between␣
↪0 and 1
    # run the scene:
    engine.run(my_scene)
```

Run the game again. This time you will see a 1000x600 window with a 800x600 area colored in light gray. The 800x600 area is accessible for the engine to draw your game contents. The engine won't be able to draw anything outside that area. The size of the area is 800x600 because that's the virtual_resolution we set when initializing the engine.

Try resizing the game window and see how the engine shrinks or stretches out the frame buffer area. As you may expect, anything your game will draw inside the area will shrink or stretch accordingly.

You have probably noticed that the engine tries to maintain the aspect ratio (width to height proportions) of the grey drawable area. We call this "adaptive stretch mode" - this is the default mode. It works like this:

```
from kaa.engine import VirtualResolutionMode
```

And then pass it when initalizing the engine:

```
with Engine(virtual_resolution=Vector(800, 600), virtual_resolution_
↪mode=VirtualResolutionMode.adaptive_stretch) as engine:
    ...
```

You can tell the engine to use the following modes when adjusting your virtual resolution to the window:

- `VirtualResolutionMode.adaptive_stretch` - the default mode. The drawable area will adapt to window size, maintaining aspect ratio and leaving black padded areas outside
- `VirtualResolutionMode.aggresive_stretch` - the drawable area will always fill the entire window - aspect ratio may not be maintained while stretching.
- `VirtualResolutionMode.no_stretch` - no stretching applied, leaving black padded areas if window is larger than virtual resolution size

---

Note: It is possible to change the virtual resolution size and mode even as the game is running.

1.1.4 Fullscreen mode

Running the game in fullscreen is very easy:

```
engine.window.fullscreen = True
```

The engine will resize the window to fit the entire screen and remove window's top bar and borders. If you select the window size manually in addition to setting fullscreen to True, the selected size will be ignored.

Kaa engine allows to alt-tab out of the game running in fullscreen.

Note: It is possible to toggle fullscreen mode and change other window properties even as the game is running.

1.1.5 End of Part 1 - full code

Feel free to experiment with window and renderer properties. Then use the following main.py content below and proceed to *Part 2 of the tutorial*

```
from kaa.engine import Engine, Scene, VirtualResolutionMode
from kaa.geometry import Vector

class MyScene(Scene):

    def update(self, dt):
        pass


with Engine(virtual_resolution=Vector(800, 600)) as engine:
    # set  window properties
    engine.window.size = Vector(800, 600)
    engine.window.title = "My first kaa game!"
    # initialize and run the scene
    my_scene = MyScene()
    engine.run(my_scene)
```

1.2 Part 2: Sprites and nodes

In a previous chapter we have covered some properties of engine, window and renderer and were able to run a game showing empty screen. Let's start drawing some actual objects in our game!

1.2.1 Loading images from files

In order to draw anything, we need to load an image file first. For this demo we will use a prepared package of assets, `available here` which includes full set of images, sounds and fonts for the tutorial. Download the file and unpack it inside the folder with main.py. You should get the following folder structure:

```
my_game/
    assets/
        gfx/
            arrow.png
            ..... other image files ....
        sfx/
            .... sound effect files ....
        music/
            .... music files ....
        fonts/
            .... font files ....
    main.py
```

Let's now load the first image (arrow.png). Add the following imports at the top of the main.py

```
from kaa.sprites import Sprite
import os
```

Then add `__init__()` method to `MyScene` and load our image there:

```
def __init__(self):
    super().__init__()
    arrow_sprite = Sprite(os.path.join('assets', 'gfx', 'arrow.png'))
```

Kaa engine loads images to objects called Sprites. With the image loaded, we can create the first few in-game objects (which will use the same Sprite).

1.2.2 Drawing objects on the screen

Object instances present on the scene are called Nodes. Let's create three arrow objects (three Nodes) using the arrow sprite.

```
from kaa.nodes import Node
```

```
def __init__(self):
    super().__init__()
    self.arrow_sprite = Sprite(os.path.join('assets', 'gfx', 'arrow.png'))
    self.arrow1 = Node(sprite=self.arrow_sprite, position=Vector(200, 200))  #␣
↪default position is Vector(0,0)
    self.arrow2 = Node(sprite=self.arrow_sprite, position=Vector(400, 300))
    self.arrow3 = Node(sprite=self.arrow_sprite, position=Vector(600, 500))
```

Run the game and... No objects are visible on the screen! It's because we created them but did not add them to the Scene. A shameful display! Let's fix it. The Scene holds a tree-like structure of Nodes, and always has the "root" Node. Let's add our objects as children of the root node:

```
def __init__(self):
    super().__init__()
    self.arrow_sprite = Sprite(os.path.join('assets', 'gfx', 'arrow.png'))
    self.arrow1 = Node(sprite=self.arrow_sprite, position=Vector(200, 200))
    self.arrow2 = Node(sprite=self.arrow_sprite, position=Vector(400, 300))
    self.arrow3 = Node(sprite=self.arrow_sprite, position=Vector(600, 500))
    self.root.add_child(self.arrow1)
    self.root.add_child(self.arrow2)
    self.root.add_child(self.arrow3)
```

Run the game again. Looks much better doesn't it? The arrows appear exactly where we put them.

Note: Sprites are immutable. Think of them as wrapper objects for image files.

1.2.3 Moving objects around

To move an object to a different position, simply set a new position:

```
def __init__(self):
    # ... previous code...
    self.arrow1.position = Vector(360, 285)
```

Run the game and check out the results!

Note: position's x and y can be floats, e.g. `arrow1.position = Vector(360.45, 285.998)` they can also be negative e.g. `arrow1.position = Vector(-50, -10)`

1.2.4 Using z-index

Hmm, arrow1 now overlaps arrow2, but what decides which one is displayed on top? Long story short: nothing decides, it is unpredictable. Let's take control by assigning objects a z-index. Object with a bigger z-index will always be rendered on top of the objects with smaller z-index.

```
def __init__(self):
    # ... previous code...
    self.arrow1.z_index = 1  # note: default z_index is 0
```

Run the game and see that arrow1 is always drawn on top of arrow2.

1.2.5 Rotating objects

To rotate an object, simply set the rotation_degrees property.

```
def __init__(self):
    # ... previous code...
    self.arrow1.rotation_degrees = 45  # note: default rotation_degrees is 0
```

Notice that you can set rotation_degrees to more than 360 degrees or to negative values.

Those more mathematically inclined can use radians. 45 degrees should be pi/ 4, right? Use `rotation` property on a node:

```
import math
self.arrow1.rotation = math.pi / 4
```

Run the game and check for yourself - arrow1 rotated 45 degrees!

1.2.6 Scaling objects

To scale an object in X or Y axis (or both), use the `scale` property. Pass a Vector object, where vector's x,y values are scaling factors for x and y axis respectively. 1 is the default scale, 2 will enlarge it twice, passing 0.5 will shrink it 50%, etc.

```
self.arrow1.scale = Vector(0.5, 1)  # note: default is Vector(1,1)
```

Re-run the game and see how X axis of the arrow was scaled down.

1.2.7 Aligning object's 'origin' (the anchor point)

Let's ask a curious question. Our 'arrow' object has spatial dimentions: 100px width and 50px height. We tell the game to draw it at some specific position e.g. (300, 200). But what does this actually mean? Which pixel of the arrow will really be drawn at position (300, 200)? The top-left pixel? Or the central pixel? Or maybe some other pixel?

By default it's the central pixel. That anchor point of a node is called 'origin'. Let's visualize the idea by drawing a 'pixel marker' image in position of arrow2 and arrow3

```
def __init__(self):
    ... previous code...
    # create pixel marker sprite
    self.pixel_marker_sprite = Sprite(os.path.join('assets', 'gfx', 'pixel-marker.png
↪'))
    # create pixel_marker 1 in the same spot as arrow2 (but with bigger z-index so we␣
↪can see it)
    self.pixel_marker1 = Node(sprite=self.pixel_marker_sprite, position=Vector(400,␣
↪300), z_index=100)
    # create pixel_marker 2 in the same spot as arrow3
    self.pixel_marker2 = Node(sprite=self.pixel_marker_sprite, position=Vector(600,␣
↪500), z_index=100)
    # add pixel markers to the scene
    self.root.add_child(self.pixel_marker1)
    self.root.add_child(self.pixel_marker2)
```

Run the game and see the markers appear on top of arrows in the central position.

Now, let's change just one thing: origin_alignment of arrow 3

```
from kaa.geometry import Alignment
```

```
def __init__(self):
    # ... previous code...
    self.arrow3.origin_alignment = Alignment.right  # default is Alignment.center
```

Re-run the game and see how arrow3 is now drawn in a different place! We did not change its position, just the origin alignment. Not surprisingly, we can see that origin marker is to the right of the node's rectangle.

You can set the origin to be in one of the 9 standard positions: top-left, top, top-right, left, central (default), right, bottom-left, bottom and bottom-right. The node's rectangular shape will be drawn according to origin position.

All transformations such as positioning, scaling or rotating are applied in relation to the origin. We'll see that in practice in the next section.

Note: What if you need a non-standard position for node's origin? You can achieve that by using two nodes in a parent - child relation. It's described in more detail in one of the next sections.

1.2.8 Updating state of objects

So far, we've been writing our code in the Scene's `__init__` method. This is a standard practice to create an initial state of the scene. Let's now try to update our objects in real-time, as the game is running!

Every scene has `update(dt)` function which will be called by the engine in a loop (with maximum frequency of 60 times per second). The `dt` parameter is an integer value how many milliseconds had passed since the last update call. You will implement most of your game logic inside the `update` function.

Let's get to it. Modify the `update` function in `MyScene` class:

```
def update(self, dt):
    #  .... previous code ....
    self.arrow2.rotation_degrees += 1  # rotating 1 degree PER FRAME (not the best␣
↪design)
    self.arrow3.rotation_degrees += 90 * dt / 1000  # rotating 90 degrees PER SECOND␣
↪(good design!)
```

Run the game and notice how the arrows rotate **around their respective origin points**. It's also worth noting that it's generally better to include dt in all formulas which transform game objects. Rotating, moving, or generally applying any other transformation by a fixed value per frame can lead to problems because it is not guaranteed that frame time (dt) will always be identical. Some frames may take longer to process than others and the visible transformations would suddenly speed up or slow down, confusing the player. Thus it's usually better to apply transformations **per second**.

1.2.9 Objects can have child objects

So far we've been adding objects (Nodes) to the root Node of the scene. But each node we create can have its own child nodes, those child nodes can have their own children and so on.

All transformations applied to a node are automatically applied to all its child nodes. Let's check this out in practice. Add the following code to the `__init__` function of the Scene.

```
def __init__(self):
    # .... previous code .....
    self.green_arrow_sprite = Sprite(os.path.join('assets', 'gfx', 'arrow-green.png'))
    self.child_arrow1 = Node(sprite=self.green_arrow_sprite, position=Vector(0,0),␣
↪rotation_degrees=90, z_index=1)
    self.arrow3.add_child(self.child_arrow1)
```

Run the game and check out the result. First thing you have probably noticed is that we set child_arrow1's position to (0,0) yet the green arrow is being shown at (600, 500)! This is because **child node's position value is not absolute but relative to the parent**. Since parent's position is (600, 500) and child's offset is (0, 0) therefore calculated

child position is (600, 500). As you have noticed the child arrow is rotating together with the parent, rotated (again, relatively) by +90 degrees.

It is very important to remember that position, scale and rotation of each node are always relative to their parent node. There is a way to get an **absolute** position, scale or rotation of a Node:

```
print(self.arrow3.absolute_position)
print(self.arrow3.absolute_rotation)
print(self.arrow3.absolute_rotation_degrees)
print(self.arrow3.absolute_scale)
```

Take some time to experiment with the parent-child system. Try changing child and parent node's properties such as position, origin_alignment, rotation, scaling etc., try updating both nodes properties inside update() function and observe the results.

Note: You can add an empty Node (without image, just `Node(position=Vector(x, y))` just to hold a position and then add a child with any desired position offset. This simple trick allows for a node to have a custom origin alignment, not limited to the 9 standard origin_alignment values.

1.2.10 Showing and hiding objects

If you need to hide or show a node, use `visible` property:

```
my_node.visible = False #  default is True
```

Hiding a node will automatically hide all its child nodes.

1.2.11 Introducing animations

So far we've been using single-frame images. Kaa engine supports frame-by-frame sprite animations. Take a look at `assets/gfx/explosion.png` file. It is a frame by frame animation of an explosion, frame size is 100x100 and there are 75 actual frames in the file.

Creating animation is a two step process:

First, we need to 'cut' each frame from the `explosion.png` file and make it a separate `Sprite`. In other words we need to have 75 Sprites, one for each frame. Fortunately we don't need to do that manually, there's a helper function for slicing spritesheets: `split_spritesheet`. Let's use it.

```
from kaa.sprites import Sprite, split_spritesheet

def __init__(self):
    # .... previous code .....
    self.explosion_spritesheet = Sprite(os.path.join('assets', 'gfx', 'explosion.png
↪')) # laod the whole spritesheet
    self.explosion_frames = split_spritesheet(self.explosion_spritesheet, frame_
↪dimensions=Vector(100,100),
        frames_count=75)  # create 75 separate <Sprite> objects
```

The function is rather self-explanatory, it takes a sprite, goes through it left to right and top to bottom, cutting out frames using specified frame dimensions. It stops after `frames_count` frames.

The second step is to create an animation, and assign it to a node. We then add the Node to the scene:

```
from kaa.transitions import NodeSpriteTransition

def __init__(self):
    # .... previous code .....
    explosion_animation = NodeSpriteTransition(self.explosion_frames, duration=1000,␣
↪loops=0)  # create animation
    self.explosion = Node(position=Vector(600, 150), transition=explosion_animation)
↪# create node
    self.root.add_child(self.explosion)  # add node to scene
```

Few things demand explanation here. First, the seemingly weird name of the animation object. Why is it called `NodeSpriteTransition`, not just `SpriteAnimation` or something similar? Why is it imported from `kaa.transitions` namespace ? The reason is because it's a part of much more general mechanism called... transitions! Transitions are recipes how node's property should evolve over time. In this case the evolving property is a sprite, but as you will see in the *Part 9 of the tutorial* there are also transitions for properties such as position, rotation, scale, color and others. The mechanism allows to 'change' those properties over time just like we change the sprite over time. That also explains why node property is called `transition`.

Let's look at the `NodeSpriteTransition` parameters. First one is obviously a list of frames, the `duration` tells how long the animation should take (in miliseconds). The `loops` parameter tells how many times the animation should repeat. 0 means infinite number of repetitions.

Run the game and behold the animated explosion, if you haven't yet!

Note: All transitions, including `NodeSpriteTransition` are immutable which means if you need to create a transition with different parameters, you need to create a new transition object. You can re-use the same transition on multiple nodes though.

Let's illustrate this on example. Let's use the same set of frames to create a new animation: longer duration, with 3 loops instead of the infinite loop, and running back and forth. Then let's add two explosion Nodes using that animation

```
def __init__(self):
    # .... previous code .....
    explosion_animation_long =  NodeSpriteTransition(self.explosion_frames,␣
↪duration=4000, loops=3,
                                                     back_and_forth=True)  # create␣
↪animation
    self.explosion2 = Node(position=Vector(100, 400), transition=explosion_animation_
↪long)
    self.explosion3 = Node(position=Vector(200, 500), transition=explosion_animation_
↪long)
    self.root.add_child(self.explosion2)
    self.root.add_child(self.explosion3)
```

Run the game and check out the new explosions. We've also learned about `back_and_forth` flag on the `NodeSpriteTransition`!

1.2.12 How to crop a Sprite

What if you want to crop the Sprite manually? Use `crop()` method on Sprite object, getting a new Sprite

```
new_sprite = self.arrow1.crop(Vector(5,5), Vector(10,20))
```

The example above will create a new, 10x20 px Sprite from arrow1, starting the crop from position (5,5).

1.2.13 Controlling animations manually

If you want to take full control of the animation you need to set each frame manually (set the `sprite` on given Node manually). It's entirely up to you how you do that, let's just say that there's something like custom transitions. We'll learn more about transitions in *Chapter 10 of the tutorual*

1.2.14 Setting a lifetime of an object

For every Node you create you can set a `lifetime` property. It is a number of miliseconds after which the node will be automatically removed from the scene. Just remember that the timer starts ticking from the moment of adding node to the scene, not from the moment of creating the Node object. If a node is already added to the Scene, the timer starts immediately.

Let's set lifetime property on one of the nodes:

```
self.explosion3.lifetime = 5000
```

Run the game, and observe that the node gets removed after 5 seconds.

1.2.15 Deleting objects from the scene

You will of course need to remove Nodes from the scene programmatically as well. It is very easy, just use the `delete()` method on the Node you wish to remove.

```
some_node.delete()
```

The node will get removed from the scene immediately. If it has child nodes, they will be removed as well, together with their child nodes and so on, recursively.

IMPORTANT: after deleting a node you should not call any of its method or access any of its properties, even the read-only properties. It will cause non deterministic efects as the game runs, eventually leading to a segmentation fault and a crash to desktop.

1.2.16 End of Part 2 - full code

We end this part of tutorial with a lot of code inside Scene's `__init__`. It starts looking messy but don't worry, we'll start the *Part 3* with a cleanup, and then we'll get to writing the actual game!

Anyway, here's the full listing of main.py after Part 2:

```
from kaa.engine import Engine, Scene
from kaa.geometry import Vector
from kaa.sprites import Sprite, split_spritesheet
from kaa.nodes import Node
from kaa.geometry import Alignment
from kaa.transitions import NodeSpriteTransition
import os

class MyScene(Scene):

    def __init__(self):
        super().__init__()
        self.arrow_sprite = Sprite(os.path.join('assets', 'gfx', 'arrow.png'))
        self.arrow1 = Node(sprite=self.arrow_sprite, position=Vector(200, 200))
```

(continues on next page)

(continued from previous page)

```
        self.arrow2 = Node(sprite=self.arrow_sprite, position=Vector(400, 300))
        self.arrow3 = Node(sprite=self.arrow_sprite, position=Vector(600, 500))
        self.root.add_child(self.arrow1)
        self.root.add_child(self.arrow2)
        self.root.add_child(self.arrow3)
        self.arrow1.position = Vector(360, 285)
        self.arrow1.z_index = 1  # note: default z_index is 0
        self.arrow1.rotation_degrees = 45
        self.arrow1.scale = Vector(0.5, 1)  # note: default is Vector(1,1)
        # create pixel marker sprite
        self.pixel_marker_sprite = Sprite(os.path.join('assets', 'gfx', 'pixel-marker.
↪png'))
        # create pixel_marker 1 in the same spot as arrow2 (but with bigger z-index␣
↪so we can see it)
        self.pixel_marker1 = Node(sprite=self.pixel_marker_sprite,␣
↪position=Vector(400, 300), z_index=100)
        # create pixel_marker 2 in the same spot as arrow3
        self.pixel_marker2 = Node(sprite=self.pixel_marker_sprite,␣
↪position=Vector(600, 500), z_index=100)
        # add pixel markers to the scene
        self.root.add_child(self.pixel_marker1)
        self.root.add_child(self.pixel_marker2)
        self.arrow3.origin_alignment = Alignment.right  # default is Alignment.center
        self.green_arrow_sprite = Sprite(os.path.join('assets', 'gfx', 'arrow-green.
↪png'))
        self.child_arrow1 = Node(sprite=self.green_arrow_sprite, position=Vector(0,0),
↪ rotation_degrees=90, z_index=1)
        self.arrow3.add_child(self.child_arrow1)
        self.explosion_spritesheet = Sprite(os.path.join('assets', 'gfx', 'explosion.
↪png')) # laod the whole spritesheet
        self.explosion_frames = split_spritesheet(self.explosion_spritesheet, frame_
↪dimensions=Vector(100,100),
            frames_count=75)  # create 75 separate <Sprite> objects
        explosion_animation = NodeSpriteTransition(self.explosion_frames,␣
↪duration=1000, loops=0)  # create animation
        self.explosion = Node(position=Vector(600, 150), transition=explosion_
↪animation)  # create node with animation
        self.root.add_child(self.explosion)  # add node to scene

        explosion_animation_long =  NodeSpriteTransition(self.explosion_frames,␣
↪duration=4000, loops=3,
                                                         back_and_forth=True)  #␣
↪create animation
        self.explosion2 = Node(position=Vector(100, 400), transition=explosion_
↪animation_long)
        self.explosion3 = Node(position=Vector(200, 500), transition=explosion_
↪animation_long)
        self.root.add_child(self.explosion2)
        self.root.add_child(self.explosion3)
        self.explosion3.lifetime = 5000


    def update(self, dt):
        #  .... previous code ....
        self.arrow2.rotation_degrees += 1  # rotating 1 degree PER FRAME (not the␣
↪best design)
        self.arrow3.rotation_degrees += 90 * dt / 1000  # rotating 90 degrees PER␣
↪SECOND (good design!)
```

(continues on next page)

(continued from previous page)

```
with Engine(virtual_resolution=Vector(800, 600)) as engine:
    # set  window properties
    engine.window.size = Vector(800, 600)
    engine.window.title = "My first kaa game!"
    # initialize and run the scene
    my_scene = MyScene()
    engine.run(my_scene)
```

1.3 Part 3: Organizing the game code

We've learned how to add objects to the nodes tree (draw them on the screen), how to transform them (move, rotate, scale), add child nodes to other nodes and how to use animations. Let's start writing the actual game!

The game will be a top-down shooter with 3 weapons: machine gun, grenade launcher and force gun (will shoot non-lethal bullets which will push enemies away) and one type of enemy (a zombie). Enemies will have a basic AI with two behavior patterns: walk towards the player or just walk towards randomly selected point. We will implement some animations such as explosions and blood splatters. We'll use kaa's physics system to detect collisions between bullets and enemies as well as between characters in the game (player and enemies). We'll add some sound effects and music for a better experience. We will also learn how to draw text and how to control a camera. Finally, we'll learn how to add more scenes, such as main screen or pause screen and how to switch between them.

It would not look good if we put all that stuff in main.py, so let's create a better structure for the game files and folders first. We'll also clean up the code we wrote before.

1.3.1 Before we begin

From this point on we're writing the actual game and the tutorial will have a lot of code in form of snippets.

Be aware that there will be two types of code examples:

A general example that explains a mechanism existing in the code. You don't need to put that code anywhere.

```
def foo()
    print('Hello world')
```

An actual code of the game we're creating. Those code snippets will have a blue header bar telling you which file you should put the code in. For example, this code should be put in `folder/subfolder/foo.py`

Listing 1: folder/subfolder/foo.py

```
def bar():
    print('hello sailor!')
```

1.3.2 Structure of directories & files

Let's start with creating the proper folders and files hierarchy structure for our game. Create the following python packages and files structure.

```
my_game/
    assets/
        ... all assets folder content here...
    common/
        __init__.py
        enums.py
    controllers/
        __init__.py
        assets_controller.py
        enemies_controller.py
        explosions_controller.py
        collisions_controller.py
        player_controller.py
    objects/
        weapons/
            __init__.py
            base.py
            force_gun.py
            grenade_launcher.py
            machine_gun.py
        bullets/
            __init__.py
            force_gun_bullet.py
            grenade_launcher_bullet.py
            machine_gun_bullet.py
        __init__.py
        player.py
        enemy.py
        explosion.py
    scenes/
        __init__.py
        gameplay.py
        pause.py
        title_screen.py
    main.py
    registry.py
    settings.py
```

Controllers package will store classes to handle the game logic and manage objects.

Objects package will hold classes for different types of objects that will appear in the game. To keep it clean we'll have one .py file for one object type.

Scenes package will hold scenes. Yes, our game will eventually have many scenes, we will get there later in the tutorial.

settings.py is a config file for our game

registry.py is a module to store global variables - we'll be able to import them from anywhere in our code.

Note: The organization above is just a suggestion, not some rigid convention required by the kaa engine. You can work out your own patterns for organizing the game files and folders, and use whatever works best for you. You don't need to follow naming conventions used in this tutorial. You can call controllers 'managers', or re-name the entry module main.py to something else. Whatever works for you.

1.3.3 Storing global variables and objects

Let's start with settings.py:

Listing 2: settings.py

```
# Let's use full HD as a base resolution for our game!
VIEWPORT_WIDTH = 1920
VIEWPORT_HEIGHT = 1080
```

Then registry.py:

Listing 3: registry.py

```
class Registry: # serious name, to look like a pro. In fact won't do anything - will␣
↪just serve as a bag for objects :))
    pass

global_controllers = Registry()
scenes = Registry()
```

1.3.4 Keep scenes in separate .py files

Let's create a stub of a Gameplay scene in `scenes/gameplay.py`

Listing 4: scenes/gameplay.py

```
from kaa.engine import Scene

class GameplayScene(Scene):

    def __init__(self):
        super().__init__()

    def update(self, dt):
        pass
```

1.3.5 Keep the main.py clean

Finally, let's now clean up the main.py. Generally, the main module should have as little lines as possible because we want the entire game logic to be in controllers, objects and scenes classes.

Listing 5: main.py

```
from kaa.engine import Engine
from kaa.geometry import Vector
import settings
from scenes.gameplay import GameplayScene

with Engine(virtual_resolution=Vector(settings.VIEWPORT_WIDTH, settings.VIEWPORT_
↪HEIGHT)) as engine:
    # set window to fullscreen mode
    engine.window.fullscreen = True
    # initialize and run the scene
```

(continues on next page)

(continued from previous page)

```
    gameplay_scene = GameplayScene()
    engine.run(gameplay_scene)
```

Our main.py looks very pro now! Run the game to make sure it works. You should see an empty, black screen. Press Alt+F4 to close it.

1.3.6 Load assets just once, from one place, and make them visible from everywhere

Proper assets management is very important. In Part 2 of the tutorial we have created Sprite objects inside Scene's __init__. It might work OK in a small game, but in the long run it's not a good idea because some scenes can be destroyed and created again. If we load assets inside scene's __init__ - we would re-load the same assets files from disk each time scene is reset (e.g. when player starts a new game).

Scene's `__init__` should only create Nodes needed to initialize the scene. Sprites and other assets-related objects are immutable, so should be created only once, when the game starts. That's what our `AssetsController` class is for. Let's edit the `assets_controller.py` file:

Listing 6: controllers/assets_controller.py

```
import os
from kaa.sprites import Sprite, split_spritesheet
from kaa.geometry import Vector

class AssetsController:

    def __init__(self):
        # Load images:
        self.background_img = Sprite(os.path.join('assets', 'gfx', 'background.png'))
        self.title_screen_background_img = Sprite(os.path.join('assets', 'gfx',
↪'title-screen.png'))
        self.player_img = Sprite(os.path.join('assets', 'gfx', 'player.png'))
        self.machine_gun_img = Sprite(os.path.join('assets', 'gfx', 'machine-gun.png
↪'))
        self.force_gun_img = Sprite(os.path.join('assets', 'gfx', 'force-gun.png'))
        self.grenade_launcher_img = Sprite(os.path.join('assets', 'gfx', 'grenade-
↪launcher.png'))
        self.machine_gun_bullet_img = Sprite(os.path.join('assets', 'gfx', 'machine-
↪gun-bullet.png'))
        self.force_gun_bullet_img = Sprite(os.path.join('assets', 'gfx', 'force-gun-
↪bullet.png'))
        self.grenade_launcher_bullet_img = Sprite(os.path.join('assets', 'gfx',
↪'grenade-launcher-bullet.png'))
        self.enemy_stagger_img = Sprite(os.path.join('assets', 'gfx', 'enemy-stagger.
↪png'))
        # few variants of bloodstains, put them in the same list so we can pick them␣
↪randomly later
        self.bloodstain_imgs = [Sprite(os.path.join('assets', 'gfx', f'bloodstain{i}.
↪png')) for i in range(1, 5)]

        # Load spritesheets
        self.enemy_spritesheet = Sprite(os.path.join('assets', 'gfx', 'enemy.png'))
        self.blood_splatter_spritesheet = Sprite(os.path.join('assets', 'gfx', 'blood-
↪splatter.png'))
        self.explosion_spritesheet = Sprite(os.path.join('assets', 'gfx', 'explosion.
↪png'))
```

(continues on next page)

(continued from previous page)

```
        # enemy-death.png has a few death animations, so make this a list
        self.enemy_death_spritesheet = Sprite(os.path.join('assets','gfx','enemy-
↪death.png'))

        # use the spritesheets to create framesets
        self.enemy_frames = split_spritesheet(self.enemy_spritesheet, frame_
↪dimensions=Vector(33, 74))
        self.blood_splatter_frames = split_spritesheet(self.blood_splatter_
↪spritesheet, frame_dimensions=Vector(50, 50))
        self.explosion_frames = split_spritesheet(self.explosion_spritesheet, frame_
↪dimensions=Vector(100, 100), frames_count=75)

        self.enemy_death_frames = [
            split_spritesheet(self.enemy_death_spritesheet.crop(Vector(0, i*74),␣
↪Vector(103*9, 74)),
                              frame_dimensions=Vector(103, 74)) for i in range(0, 5)
        ]
```

The code is using features we've learned in previous chapter: creating a new Sprite, using crop method and using split_spritesheet to prepare individual animation frames which we'll use later.

Feel free to review the contents of the `assets/gfx` folder to verify we're loading the files correctly.

As stated above, we want the assets controller to initialize just once and then be globally visible. Let's modify the `main.py` in a following way:

Listing 7: main.py

```
import registry
from controllers.assets_controller import AssetsController

with Engine(virtual_resolution=Vector(settings.VIEWPORT_WIDTH, settings.VIEWPORT_
↪HEIGHT)) as engine:
    # initialize global controllers and keep them in the registry
    registry.global_controllers.assets_controller = AssetsController()
    # ..... rest of the code .....
```

1.3.7 It's good to keep scenes in a global registry too

It's practical to store scene instances in the global registry as well. That will make them accessible from anywhere in the code. Let's modify that part of `main.py` where GameplayScene is created:

Listing 8: main.py

```
with Engine(virtual_resolution=Vector(settings.VIEWPORT_WIDTH, settings.VIEWPORT_
↪HEIGHT)) as engine:
    # ..... previous code .....
    # initialize scenes and keep them in the registry
    registry.scenes.gameplay_scene = GameplayScene()
    engine.run(registry.scenes.gameplay_scene)
```

1.3.8 Write classes for your in-game objects and inherit from kaa.Node

It would look much better if we could add a <Player> object to a scene, not just some generic <Node>, right? Let's do this.

Let's write a Player class that extends kaa Node. <Player> instance will represent a character controlled by the player.

Listing 9: objects/player.py

```
from kaa.nodes import Node
import registry


class Player(Node):

    def __init__(self, position, hp=100):
        # node's properties
        super().__init__(z_index=10, sprite=registry.global_controllers.assets_
↪controller.player_img, position=position)
        # custom properties
        self.hp = hp
        self.current_weapon = None
```

By extending Node we can introduce our custom properties, such as player's hit points. Also, notice how we imported and used our `registry.py` to access the sprite stored in the assets controller.

Let's create classes for weapons the same way. They won't have any custom properties for now. We'll have a base class, called WeaponBase extending Node, and all our wepons will then extend the WeaponBase.

Listing 10: objects/weapons/base.py

```
from kaa.nodes import Node


class WeaponBase(Node):

    def __init__(self, *args, **kwargs):
        super().__init__(z_index=20, *args, **kwargs)
```

Listing 11: objects/weapons/machine_gun.py

```
import registry
from objects.weapons.base import WeaponBase


```

(continues on next page)

(continued from previous page)

```
class MachineGun(WeaponBase):

    def __init__(self):
        # node's properties
        super().__init__(sprite=registry.global_controllers.assets_controller.machine_
↪gun_img)
```

Listing 12: objects/weapons/force_gun.py

```
import registry
from objects.weapons.base import WeaponBase


class ForceGun(WeaponBase):

    def __init__(self):
        # node's properties
        super().__init__(sprite=registry.global_controllers.assets_controller.force_
↪gun_img)
```

Listing 13: objects/weapons/grenade_launcher.py

```
import registry
from objects.weapons.base import WeaponBase


class GrenadeLauncher(WeaponBase):

    def __init__(self):
        # node's properties
        super().__init__(sprite=registry.global_controllers.assets_controller.grenade_
↪launcher_img)
```

1.3.9 Implement object-related logic inside object classes

We need Player to hold a weapon. Let's implement a `change_weapon` method in the `Player` class. This method will be responsible for putting weapon into player's hands :) or speaking more technically: it will replace weapon's Node (which will be Player's child node) with a new one and remember currently selected weapon.

To hide the internals, we want the caller to only pass a simple enumerated value indicating new weapon, like so:

```
player.change_weapon(WeaponType.GrenadeLauncher)
```

Let's create weapon types enum first:

Listing 14: common/enums.py

```
import enum


class WeaponType(enum.Enum):
    MachineGun = 1
    GrenadeLauncher = 2
    ForceGun = 3
```

And then add the change_weapon method in the `Player` class:

Listing 15: objects/player.py

```
from kaa.geometry import Vector
from common.enums import WeaponType
from objects.weapons.force_gun import ForceGun
from objects.weapons.grenade_launcher import GrenadeLauncher
from objects.weapons.machine_gun import MachineGun

class Player(Node):

    def change_weapon(self, new_weapon):
        if self.current_weapon is not None:
            self.current_weapon.delete()  # delete the weapon's node from the scene
        if new_weapon == WeaponType.MachineGun:
            weapon = MachineGun()  # position relative to the Player
        elif new_weapon == WeaponType.GrenadeLauncher:
            weapon = GrenadeLauncher()
        elif new_weapon == WeaponType.ForceGun:
            weapon = ForceGun()
        else:
            raise Exception('Unknown weapon type: {}'.format(new_weapon))
        self.add_child(weapon)  # add the weapon node as player's child node (to make␣
↪the weapon move and rotate together with the player)
        self.current_weapon = weapon  # remember the current weapon
```

Let's make the player start with machine gun. Add this line at the end of `Player`'s `__init__`:

Listing 16: objects/player.py

```
self.change_weapon(WeaponType.MachineGun)
```

1.3.10 Implement higher-tier logic in controller classes

Let's now write a controller class to manage the Player. Generally we want the controller classes to be used for higher-tier logic such as interactions between in-game objects and other classes (controllers or other in-game objects), managing collections, handling input, and so on...

Another important thing we want controllers to do is to add initial objects to the scene. Let's start with exactly that:

Listing 17: controllers/player_controller.py

```
import settings
from objects.player import Player
from kaa.geometry import Vector

class PlayerController:

    def __init__(self, scene):
        self.scene = scene
        self.player = Player(position=Vector(settings.VIEWPORT_WIDTH/2, settings.
↪VIEWPORT_HEIGHT/2))
        self.scene.root.add_child(self.player)
```

Note: As your code base will grow and you'll add more objects and controllers you will sometimes face a dillema

where to put your code: in the object class, in the controller class or maybe even directly in the scene class? We can't give you precise answers here, just use common sense and general good programming practices for keeping your code clean.

Let's add the player controller to the scene:

Listing 18: scenes/gameplay.py

```
from controllers.player_controller import PlayerController


class GameplayScene(Scene):

    def __init__(self):
        super().__init__()
        self.player_controller = PlayerController(self)
```

Finally, let's run the game! We should see the player in the middle of the screen, holding the machine gun.

Lastly, let's add some nicer background (black background is not fun).

Listing 19: scenes/gameplay.py

```
import registry
import settings
from kaa.nodes import Node
from kaa.geometry import Vector
# ... other imports...

class GameplayScene(Scene):

    def __init__(self):
        super().__init__()
        self.root.add_child(Node(sprite=registry.global_controllers.assets_controller.
↪background_img,
                                 position=Vector(settings.VIEWPORT_WIDTH/2, settings.
↪VIEWPORT_HEIGHT/2),
                                 z_index=0))
        # .... rest of the function ....
```

Run the game and enjoy the sights.

Let's move on to the *Part 4 of the tutorial* where we'll learn how to handle input from mouse and keyboard.

1.4 Part 4: Handling Input

We have our hero drawn on the screen, holding a machine gun. In this chapter we will implement the following input-related stuff:

- move our hero around with WSAD keys,
- cycle weapons by pressing tab key
- switch to selected weapon by pressing 1,2 and 3
- look around by moving the mouse
- shoot by pressing left mouse button.

The best place to handle input is `update(dt)` function. But we don't want to put everything in the scene's `update(dt)` as the code would grow too large. Let's add an `update(dt)` function to `PlayerController` class:

Listing 20: controllers/player_controller.py

```
class PlayerController:

# .... rest of the class .....

def update(self, dt):
    pass
```

Then let's call that method from the `GameplayScene`:

Listing 21: scenes/gameplay.py

```
class GameplayScene(Scene):

    # ..... rest of the class .........

    def update(self, dt):
        self.player_controller.update(dt)

        #....... rest of the method .........
```

1.4.1 Three ways for handling input

Kaa offers three ways for handling input.

- You can actively check given key/button status (pressed, released etc.). You do that by calling appropriate methods:
  - Scene's `input.keyboard` methods for keyboard (e.g. `input.keyboard.is_pressed(KeyCode.esc)`)
  - Scene's `input.mouse` methods for mouse (e.g. `input.keyboard.is_pressed(MouseButton.left)`)
  - Scene's `input.controller` methods for controllers (e.g. `input.controller.is_pressed(ControllerButton.a)`)
  - Scene's `input.system` methods for system (e.g. `system.get_clipboard_text()`)
- You can iterate through events returned by the Scene's `input.events()` method to check for input events (keystrokes, mouse clicks, controller sticks moved, etc.).
- You can subscribe to specific type of events using Scene's `input.register_callback()` method.

We're going to use the first two methods in the tutorial.

1.4.2 Handling input from keyboard

The function which you can call at any time and get an answer if a keyboard key is up or down is `input.keyboard.is_pressed()`. Let's use it in our `player_controller.py`:

Listing 22: controllers/player_controller.py

```
from kaa.input import Keycode

class PlayerController:

    # .... rest of the class .....

    def update(self, dt):
        if self.scene.input.keyboard.is_pressed(Keycode.w):
            self.player.position += Vector(0, -3)
        if self.scene.input.keyboard.is_pressed(Keycode.s):
            self.player.position += Vector(0, 3)
        if self.scene.input.keyboard.is_pressed(Keycode.a):
            self.player.position += Vector(-3, 0)
        if self.scene.input.keyboard.is_pressed(Keycode.d):
            self.player.position += Vector(3, 0)
```

Run the game and see how our hero can now move using WSAD keys!

Note: To check if a key is in "released" state use `scene.input.keyboard.is_released()`

But hey, wasn't something like this an example of a bad practice? We just hardcoded hero's speed to 3 pixels (actually: 3 units of virtual resolution) per frame, ignoring the dt value! It means if the dt is 15 miliseconds the hero will move the same distance as when the frame takes 10 times longer (dt is 150 miliseconds). Also, shouldn't hero speed value be defined in settings.py and imported from there rather just put directly in the code like some "magic number"?

Yup, those are all valid points. Don't worry - we'll refactor that code later, when we start working with the physics.

Let's now implement a function to cycle through weapons. Add the following code to the `Player` class:

Listing 23: controllers/player.py

```
class Player(Node):

    # .... rest of the class ....

    def cycle_weapons(self):
        if self.current_weapon is None:
            return
        elif isinstance(self.current_weapon, MachineGun):
            self.change_weapon(WeaponType.GrenadeLauncher)
        elif isinstance(self.current_weapon, GrenadeLauncher):
            self.change_weapon(WeaponType.ForceGun)
        elif isinstance(self.current_weapon, ForceGun):
            self.change_weapon(WeaponType.MachineGun)
```

Pretty self explanatory. Now let's try calling this function when tab key is pressed. Append the following code to the `update()` function in `PlayerController`:

Listing 24: controllers/player_controller.py

```
class PlayerController:

    # .... rest of the class .....
```

(continues on next page)

(continued from previous page)

```
    def update(self, dt):
        # ....... rest of the function ........
        if self.scene.input.keyboard.is_pressed(Keycode.tab):
            self.player.cycle_weapons()
```

Run the game and press tab.... whoa!!! It makes our hero change weapons so fast! This is because the `update()` function is called by the engine as frequently as 60 times per second, so our `cycle_weapons()` function is called 60 times per second (as long as the tab key is pressed).

Let's fix this! There is another method of handling input from keyboard, it captures individual key strokes.

1.4.3 Handling events from keyboard

Let's remove the `if self.scene.input.keyboard.is_pressed(Keycode.tab):` part from the update function inside `PlayerController` and put the following code instead:

Listing 25: controllers/player_controller.py

```
from common.enums import WeaponType

class PlayerController:

    # ..... rest of the class ........

    def update(self, dt):

        # ....... rest of the method .........

        for event in self.scene.input.events(): # iterate over all events which␣
↪occurred during this frame
            if event.keyboard_key:  # check if the event is a keyboard key related␣
↪event
                if event.keyboard_key.is_key_down:  # check if the event is "key down␣
↪event"
                    # check which key was pressed down:
                    if event.keyboard_key.key == Keycode.tab:
                        self.player.cycle_weapons()
                    elif event.keyboard_key.key == Keycode.num_1:
                        self.player.change_weapon(WeaponType.MachineGun)
                    elif event.keyboard_key.key == Keycode.num_2:
                        self.player.change_weapon(WeaponType.GrenadeLauncher)
                    elif event.keyboard_key.key == Keycode.num_3:
                        self.player.change_weapon(WeaponType.ForceGun)
```

Run the game. Works much better now, right?

Let's take a look at the code. What happens here is we iterate on all events which occurred during current frame. Each Event object has identical structure - it holds properties such as `keyboard_key`, `mouse_button` and about a dozen others. Of those properties only one will be non null - that indicates what type of event this is. For example, if the `keyboard_key` property is not `None` it means this event is a keyboard key related event. Accessing `keyboard_key` property gives access to new properties, specific for this type of event. Refer to *input.Event* documentation for more details.

In our case the `keyboard_key.is_key_down` event is published on a first key stroke. That allows us to react to individual key stroke events more naturally, unlike checking for a "key pressed" status 60 times a second.

Note: You can use `event.keyboard_key.is_key_up` to detect when a key was released.

We now have ability to move our hero, cycle through weapons with tab, and select weapon with 1, 2 and 3.

One more thing before we move on, it's annoying to press ALT+F4 to close the window, let's just bind it with pressing 'q'. Let's update the `update()` (no pun intended) in the Scene.

Listing 26: scenes/gameplay.py

```
from kaa.input import Keycode

class GameplayScene(Scene):

    # ....... rest of the class ...........

    def update(self, dt):
        self.player_controller.update(dt)

        for event in self.input.events():
            if event.keyboard_key:  # check if the event is a keyboard key related␣
↪event
                if event.keyboard_key.is_key_down:  # check if the event is "key down␣
↪event"
                    # check which key was pressed down:
                    if event.keyboard_key.key == Keycode.q:
                        self.engine.quit()
```

1.4.4 Getting mouse position

Getting mouse position is very easy. All we need is to call `input.mouse.get_position()` on our scene instance.

Let's get current mouse position and use it to rotate the player towards the mouse pointer.

Listing 27: controllers/player_controller.py

```
class PlayerController:

    # ..... rest of the class ........

    def update(self, dt):

        # ....... rest of the method .........

        mouse_pos = self.scene.input.mouse.get_position()
        player_rotation_vector = mouse_pos - self.player.position
        self.player.rotation_degrees = player_rotation_vector.to_angle_degrees()
```

Let's look at the code: to get a direction vector between positions A and B we need to substract those two vectors. We then use `to_angle_degrees()` on a vector to get a number between 0 and 360 representing vector's angle. Finally we set player's rotation (in degrees) to the calculated value.

Run the game. We can now walk with WSAD, change weapons with tab, 1, 2, and 3 keys, and we can aim! It starts looking good! Let's now add a shooting mechanics!

1.4.5 Getting mouse button click events

Handling mouse click events, is very similar to handling keyboard events. We can actively check if mouse button is pressed/released or we can check for mouse button events present in the Scene's `input.events()` list.

Look at the example below but don't add it to the game's code yet. We'll do that in the next chapter.

```
from kaa.input import MouseButton

# active check if mouse key is pressed:
if scene.input.mouse.is_pressed(MouseButton.left):
    # ..... do stuff ....

for event in self.scene.input.events():
    # check if it's a mouse button - related event and if it's about mouse button␣
↪being pressed:
    if event.mouse_button and event.mouse_button.is_button_down:
        # check which button the event is about:
        if event.mouse_button.button == MouseButton.right:
            # ..... do stuff .....
```

We will use the left mouse button click in the *next part of the tutorial*, where we'll implement shooting and collision handling.

1.5 Part 5: Physics

In this chapter we will implement physics in the game. We will add enemies and also implement shooting at them with 3 weapons:

- Machine gun - will shoot regular bullets, which will deal damage to enemies they hit.
- Grenade launcher - grenades will explode on collision (triggering already known explosion animation) and deal damage to enemies in certain radius and apply force pushing enemies away
- Force gun - will shoot a large bullets which won't do any damage, just freely interact with enemies and with each other

1.5.1 Understanding SpaceNode, BodyNode and HitboxNode

We need to learn about 3 new types of nodes which we need to simulate the physics in the game:

- `SpaceNode` - it represents physical simulation environment. Typically, a scene will need just one SpaceNode, but you can have more if needed. SpaceNode has the following properties:
  - `gravity` - a Vector. A force affecting all BodyNodes added to that SpaceNode. Default is zero vector (no gravity).
  - `damping` - a float between 0 and 1, representing friction forces in the simulation space. The smaller it is, the faster a freely moving objects will slow down. Default is 1 (no damping)
- `BodyNode` - represents a physical body. It has the same properties as Node (in fact it inherits from the Node class) but adds a few new ones, such as:
  - `body_type` - enum value, determining the type of the body node, check out *types of body nodes*
  - `mass` - a float, heavier objects will hit harder :)

  - `velocity` - a Vector. Vector's rotation is objects' movement direction and vector's length is how fast the object is moving. Default is zero vector (no velocity).
  - `angular_velocity` - a float. How fast the object is rotating around its center. Positive and negative values represent clockwise and anti-clockwise rotation speed respectively. Default is zero.
  - `force` - a Vector, representing a force working on the object. The force vector is reset to zero on each frame. Non-zero force applied each frame will cause the object to accelerate. Default is zero vector (no force).
  - and few others (out of scope of the tutorial, check out the API reference on *`physics.BodyNode`* for more info)
- `HitboxNode` - represents an area of a BodyNode which can collide with other HitboxNodes. A BodyNode can have multiple HitboxNodes. A BodyNode without HitboxNodes has all physical properties calculated normally but won't collide with anything! HitboxNode properties include:
  - `shape` - defines a shape of the hitbox, must be an instance of `kaa.geometry.Circle` or `kaa.geometry.Polygon`
  - `mask` - user-defined enum.IntFlag, indicating "what type of object I am"
  - `collision_mask` - user-defined enum.IntFlag, indicating "what type(s) of objects I can collide with"
  - `trigger_id` - a user-defined ID used for collision handler function

When working with regular Nodes, we could build any tree-like structures we wanted, with multiple levels of nested Nodes. When working with physical Nodes some restrictions apply:

- BodyNode must be a child of a SpaceNode. It cannot be a child of other node type.
- BodyNode cannot have other BodyNodes as children. It can have regular Nodes as children though.
- HitboxNode must be a child of BodyNode. It cannot be a child of any other node type. BodyNode can have any number of HitboxNodes (including zero).
- HitboxNode cannot have another HitboxNode as a child, it can have regular Node children though.

Note: Space node doesn't have to be a child of a root node, in fact it can be anywhere in the node tree but for clarity it's recommended to have it as a direct child of the root node.

Note: Physics engine available in kaa is a wrapper for an excellent 2D physics engine written in C++ named Chipmunk. Kaa surfaces a lot of Chipmunk methods and properties, but not all yet. New features are coming soon!

1.5.2 Why a BodyNode cannot have other BodyNodes as children ?

As you'll work on more complex games you'll notice that the most significant restriction is that BodyNode cannot have other BodyNodes as children. It means we cannot have a tree-like structure of colliding objects, the list of physical objects in the scene must be a flat list!. It may seem like a serious constraint, but there are good reasons for it. The purpose of physics engine is to calculate object's position, rotation, velocity etc. based on environment properties (gravity, damping) and interactions (e.g. collisions) with other dynamic objects. A node whose transformations (position, rotation) would be calculated in relation to its parent, regardless of the physical environment (like it is with regular Nodes) simply stops being a physical node and becomes just a picture drawn on the screen.

Having said that, there are ways in which you can simulate a more complex or hierarchical structure of physical objects

- Apply all BodyNode transformations manually. In other words do the calculations on your own and set the object's position and/or rotation manually.

- Spatial queries - it allows to programatically ask a question like "here's a polygon (circle, segment), tell me which HitboxNodes/BodyNodes it collides with"
- Joints - this feature is to be implemented next. You will be able to connect BodyNodes with 'joints' and they will work together.

1.5.3 Types of BodyNodes

A `BodyNode` can be one of three types. This is determined by setting `body_type` property on a `BodyNode`.

- static (`kaa.physics.BodyNodeType.static`) - this node cannot change position or rotation. Basically a performance hint for the engine. Useful for non-moving platforms, walls etc.
- kinematic (`kaa.physics.BodyNodeType.kinematic`) - the node can move but does not have a mass (you can set the mass but it won't change its behavior) therefore no environmental effects (such as damping or gravity) can affect it. When colliding with other objects it will behave as a static object. Using kinemtaic bodies is useful when you're interested just in detecting a collision and handle all consequences on your own.
- dynamic (`kaa.physics.BodyNodeType.dynamic`) - fully dynamic node. Useful for a 'free' objects which you add to the environment and let the engine work out all the physics.

1.5.4 Implement the first BodyNode with a hitbox

Let's start using physics in our game. First let's define enum flags which we'll use to control what collides with what.

Listing 28: common/enums.py

```
class HitboxMask(enum.IntFlag):
    player = enum.auto()
    enemy = enum.auto()
    bullet = enum.auto()
    all = player | enemy | bullet
```

Next let's add a SpaceNode to the Scene - it will be a container for all BodyNodes.

Listing 29: scenes/gameplay.py

```
from kaa.physics import SpaceNode

class GameplayScene(Scene):

    def __init__(self):
        super().__init__()
        self.space = SpaceNode()
        self.root.add_child(self.space)
        self.player_controller = PlayerController(self)

    # ....... rest of the class ......
```

We also need to change the line in the `PlayerController` which adds `Player` to the scene. We shall now add the player to the space node.

Listing 30: controllers/player_controller.py

```
# inside __init__ :
self.scene.space.add_child(self.player)
```

Let's add few variables to settings.py. We'll need it later, just trust me and add that stuff for now :)

Listing 31: settings.py

```
COLLISION_TRIGGER_PLAYER = 1
COLLISION_TRIGGER_ENEMY = 2
COLLISION_TRIGGER_MG_BULLET = 3
COLLISION_TRIGGER_GRENADE_LAUNCHER_BULLET = 4
COLLISION_TRIGGER_FORCE_GUN_BULLET = 5

PLAYER_SPEED = 150
FORCE_GUN_BULLET_SPEED = 300
MACHINE_GUN_BULLET_SPEED = 1200
GRENADE_LAUNCHER_BULLET_SPEED = 200
```

Finally, let's make the `Player` object to inherit from a `BodyNode`, making it a physical object. Let's give it a mass of 1. Let's also add a hitbox node to the player!

Listing 32: objects/player.py

```
import settings
from kaa.physics import BodyNode, BodyNodeType, HitboxNode
from kaa.geometry import Vector, Polygon
from common.enums import WeaponType, HitboxMask

class Player(BodyNode): # changed from kaa.Node

    def __init__(self, position, hp=100):
        super().__init__(body_type=BodyNodeType.dynamic, mass=1,
                         z_index=10, sprite=registry.global_controllers.assets_
↪controller.player_img, position=position)
        # create a hitbox and add it as a child node to the Player
        self.add_child(HitboxNode(
            shape=Polygon([Vector(-10, -25), Vector(10, -25), Vector(10, 25), Vector(-
↪10, 25), Vector(-10, -25)]),
            mask=HitboxMask.player,
            collision_mask=HitboxMask.enemy,
            trigger_id=settings.COLLISION_TRIGGER_PLAYER
        ))
        # .......... rest of the function ...........
```

As we can see, we've added a rectangular hitbox, with mask 'player' and told the engine it should collide with hitboxes whose mask is 'enemy' - we will add those soon. We have also set a trigger_id for a hitbox (basically, a custom integer number) - the meaning of this ID will also become clear soon.

A few important remarks about Polygons of hitboxes:

- they must be convex
- Polygon's coordinates are relative to the node origin
- they don't need to be closed - the first and the last point don't have to be the same. Kaa will close them automatically.

Run the game and make sure everything works. The gameplay did not change at all, but our hero is now a physical object!

Remember the naive implementation of player movement (setting player's position on WSAD keys pressed)? From physic's engine standpoint manual change of objects position makes no sense. Let's set player's `velocity` instead, and let the physics engine calculate the position.

Listing 33: controllers/player_controller.py

```
def update(dt):
    self.player.velocity=Vector(0,0) # reset velocity to zero, if no keys are pressed␣
↪the player will stop

    if self.scene.input.keyboard.is_pressed(Keycode.w):
        self.player.velocity += Vector(0, -settings.PLAYER_SPEED)
    if self.scene.input.keyboard.is_pressed(Keycode.s):
        self.player.velocity += Vector(0, settings.PLAYER_SPEED)
    if self.scene.input.keyboard.is_pressed(Keycode.a):
        self.player.velocity += Vector(-settings.PLAYER_SPEED, 0)
    if self.scene.input.keyboard.is_pressed(Keycode.d):
        self.player.velocity += Vector(settings.PLAYER_SPEED, 0)
    # ...... rest of the function ........
```

Run the game and make sure it works. Player's position will now be calculated by the physics engine, and we don't need to worry about frame duration - it's all handled automatically by the physics engine.

1.5.5 Drawing hitboxes on the screen

Hitbox nodes are invisible by default, but sometimes it's good to see them (e.g. to check if they're positioned correctly). We can do that by setting `color` property. Using `z_index` is also advisable to make the hitbox node be drawn on top of its `BodyNode`.

```
from kaa.colors import Color

# to make hitbox node visible just set its color and a high enough z_index
hitbox_node.color = Color(1, 0, 1, 0.3)
hitbox_node.z_index = 1000
```

Feel free to experiment with setting player's hitbox color, then move on to the next section.

1.5.6 Adding more BodyNodes

We have the player with a gun in hand but where are the enemies? Let's add some. First, let's write an `Enemy` class. Just like the player, the enemy must be a `BodyNode` because we want it to be a physical object with a hitbox node attached.

Listing 34: objects/enemy.py

```
from kaa.physics import BodyNodeType, BodyNode, HitboxNode
from kaa.geometry import Vector, Polygon
from common.enums import HitboxMask
import registry
import settings
from kaa.transitions import NodeSpriteTransition
import random
```

(continues on next page)

(continued from previous page)

```
class Enemy(BodyNode):

    def __init__(self, position, hp=100, *args, **kwargs):
        # node's properties
        super().__init__(body_type=BodyNodeType.dynamic, mass=1,
                         z_index=10, position=position,
                         transition=NodeSpriteTransition(registry.global_controllers.
↪assets_controller.enemy_frames,
                                                         duration=max(200, random.
↪gauss(400,100)), loops=0),
                         *args, **kwargs)
        # create a hitbox and add it as a child node to the Enemy
        self.add_child(HitboxNode(
            shape=Polygon([Vector(-8, -19), Vector(8, -19), Vector(8, 19), Vector(-8,␣
↪19), Vector(-8, -19)]),
            mask=HitboxMask.enemy,
            collision_mask=HitboxMask.all,
            trigger_id=settings.COLLISION_TRIGGER_ENEMY,
        ))
        # custom properties
        self.hp = hp
```

We're using the already known features: creating an animation loop (using `NodeSpriteTransition`), and adding a hitbox as a child node.

Next, let's write `EnemiesController` class which will have methods such as `add_enemy` and `remove_enemy`. It will also have an `update()` function where we will implement enemies AI. We shall add some initial enemies to the scene in the `__init__`.

Listing 35: controllers/enemies_controller.py

```
import random
from objects.enemy import Enemy
from kaa.geometry import Vector

class EnemiesController:

    def __init__(self, scene):
        self.scene = scene
        self.enemies = []
        # add some initial enemies
        self.add_enemy(Enemy(position=Vector(200, 200), rotation_degrees=random.
↪randint(0, 360)))
        self.add_enemy(Enemy(position=Vector(1500, 600), rotation_degrees=random.
↪randint(0, 360)))
        self.add_enemy(Enemy(position=Vector(1000, 400), rotation_degrees=random.
↪randint(0, 360)))
        self.add_enemy(Enemy(position=Vector(1075, 420), rotation_degrees=random.
↪randint(0, 360)))
        self.add_enemy(Enemy(position=Vector(1150, 440), rotation_degrees=random.
↪randint(0, 360)))

    def add_enemy(self, enemy):
        self.enemies.append(enemy)  # add to the internal list
        self.scene.space.add_child(enemy)  # add to the scene
```

(continues on next page)

(continued from previous page)

```
    def remove_enemy(self, enemy):
        self.enemies.remove(enemy)  # remove from the internal list
        enemy.delete()  # remove from the scene

    def update(self, dt):
        pass
```

Let's put the controller in the scene and hook up the `update()`:

Listing 36: scenes/gameplay.py

```
from controllers.enemies_controller import EnemiesController

class GameplayScene(Scene):

    def __init__(self):
        # ... rest of the function ....
        self.enemies_controller = EnemiesController(self)

    def update(self, dt):
        self.player_controller.update(dt)
        self.enemies_controller.update(dt)
        # ... rest of the function
```

Run the game. We have the enemies on the scene! They're animated but not moving yet. They're regular physical objects, as you run into them they'll collide with you and with each other. Since we're not applying any forces to enemies yet it looks as if they were on an ice rink :)

Let's add a feature of spawning enemies by pressing SPACE. The enemy shall be spawned at current mouse pointer position.

Listing 37: controllers/player_controller.py

```
import random
from objects.enemy import Enemy

class PlayerController:

    def update(self, dt):
        # .... rest of the function
        for event in self.scene.input.events():
            if event.keyboard_key:
                # ... other keyboard events ....
                elif event.keyboard_key.key == Keycode.space:
                    self.scene.enemies_controller.add_enemy(Enemy(position=self.scene.
↪input.mouse.get_position(),
                        rotation_degrees=random.randint(0,360)))
```

Run the game and see how you can spawn them by pressing space bar! Cool isn't it?

You can take a moment to make some experiments, for instance:

- try setting `damping` on the `SpaceNode` (in scenes/gameplay.py) to a very low value e.g. 0.01 and see how it works! Values greater than 1 will result in a funny effect of objects accelerating just by moving in the environment.
- try giving enemies different masses (e.g. randomly) and observe how it affects them as they collide with each other.

We now know everything to implement shooting the Force Gun - it will basically shoot a dynamic BodyNode objects which will collide with enemies, player and with each other. We're going to give those nodes a lifetime of 10 seconds.

Let's implement the bullet object first. It's going to be really simple: a BodyNode with a random mass, a circular hitbox and a lifetime of 10 seconds.

Listing 38: objects/bullets/force_gun_bullet.py

```
import random
from kaa.physics import BodyNode, BodyNodeType, HitboxNode
from kaa.geometry import Circle
import registry
import settings
from common.enums import HitboxMask


class ForceGunBullet(BodyNode):

    def __init__(self, *args, **kwargs):
        super().__init__(sprite=registry.global_controllers.assets_controller.force_
↪gun_bullet_img,
                         z_index=30,
                         body_type=BodyNodeType.dynamic,
                         mass=random.uniform(0.5, 8),  # a random mass,
                         lifetime=10000, # will be removed from the scene␣
↪automatically after 10 secs
                         *args, **kwargs)
        self.add_child(HitboxNode(shape=Circle(radius=10),
                                  mask=HitboxMask.bullet,
                                  collision_mask=HitboxMask.all,
                                  trigger_id=settings.COLLISION_TRIGGER_FORCE_GUN_
↪BULLET))
```

Next, let's add methods for shooting in the `WeaponBase` class and in the `ForceGun` class:

Listing 39: objects/weapons/base.py

```
from kaa.nodes import Node
from kaa.geometry import Vector


class WeaponBase(Node):

    def __init__(self, *args, **kwargs):
        super().__init__(z_index=20, *args, **kwargs)
        self.cooldown_time_remaining = 0

    def shoot_bullet(self):
        raise NotImplementedError  # must be implemented in the derived class

    def get_cooldown_time(self):
        raise NotImplementedError  # must be implemented in the derived class

    def get_initial_bullet_position(self):
        player_pos = self.parent.position
        player_rotation = self.parent.rotation_degrees
        weapon_length = 50  # the bullet won't originate in the center of the player␣
↪position but 50 pixels from it
```

(continues on next page)

(continued from previous page)

```
        result = player_pos + Vector.from_angle_degrees(player_rotation).
↪normalize()*weapon_length
        return result
```

Listing 40: objects/weapons/force_gun.py

```
import registry
import settings
from objects.bullets.force_gun_bullet import ForceGunBullet
from objects.weapons.base import WeaponBase
from kaa.geometry import Vector

class ForceGun(WeaponBase):

    def __init__(self):
        # node's properties
        super().__init__(sprite=registry.global_controllers.assets_controller.force_
↪gun_img)

    def shoot_bullet(self):
        bullet_position = self.get_initial_bullet_position()
        bullet_velocity = Vector.from_angle_degrees(self.parent.rotation_degrees) *␣
↪settings.FORCE_GUN_BULLET_SPEED
        self.scene.space.add_child(ForceGunBullet(position=bullet_position,␣
↪velocity=bullet_velocity))
        # reset cooldown time
        self.cooldown_time_remaining =  self.get_cooldown_time()

    def get_cooldown_time(self):
        return 0.250
```

The maths in the `shoot_bullet` and `get_initial_bullet_position` is fairly simple, but let's highlight a few things here. `get_initial_bullet_position` basically returns a player's position offset by 50 pixels towards the direction where the player is rotated (where he points his gun). This way the bullet will spawn at the end of the weapon's barrel. Spawning it in the center of the player would not look good! We're using Vector's method `from_angle_degrees` to create a normal (length of 1) vector rotated in the direction of the player, multiply by 50 and add player position. `shoot_bullet` is even easier, it just adds a bullet velocity, again, creating vector rotated at direction where player is pointing his gun and then multiplying by bullet speed. Finally we set the cooldown time to weapon's value.

The last thing is to wire it all up in the `PlayerController` inside the `update()` function:

Listing 41: controllers/player_controller.py

```
from kaa.input import Keycode, MouseButton

class PlayerController:
    # .... rest of the class ....

    def update(self, dt):
        # .... rest of the function ....

        # Handle weapon logic
        if self.player.current_weapon is not None:
            # decrease weapons cooldown time by dt
            self.player.current_weapon.cooldown_time_remaining -= dt
```

(continues on next page)

(continued from previous page)

```
            # if left mouse button pressed and weapon is ready to shoot, then, well,␣
↪shoot a bullet!
            if self.scene.input.mouse.is_pressed(MouseButton.left) and self.player.
↪current_weapon.cooldown_time_remaining<0:
                self.player.current_weapon.shoot_bullet()
```

Run the game! You can now shoot them with the force gun! How cool is it?

Did you get `NotImplementedError`? It's because other weapons are not implemented, just look at the code! Change to ForceGun by pressing 3 and then try shooting. Better? Much better!

The game slowly starts looking like a playable thing. We can move around, spawn enemies and shoot our Force Gun at them.

Let's now do shooting the machine gun!

1.5.7 Kinematic BodyNodes

Let's start with the machine gun bullet object. It's similar to Force Gun bullet but will use different sprite and will have a rectangular hitbox that collides only with enemies.

The most important difference though is that we'll make it a kinematic body type. As said before this body type is useful when we want to handle collisions entirely on our own.

First let's add the machine gun bullet object and implement shooting logic:

Listing 42: objects/bullets/machine_gun_bullet.py

```
import random
import registry
import settings
from kaa.physics import BodyNode, BodyNodeType, HitboxNode
from kaa.geometry import Polygon, Vector
from common.enums import HitboxMask


class MachineGunBullet(BodyNode):

    def __init__(self, *args, **kwargs):
        super().__init__(sprite=registry.global_controllers.assets_controller.machine_
↪gun_bullet_img,
                         z_index=30,
                         body_type=BodyNodeType.kinematic, # MG bullets are kinematic␣
↪bodies
                         lifetime=3000, # will be removed from the scene␣
↪automatically after 3 secs
                         *args, **kwargs)
        self.add_child(HitboxNode(shape=Polygon([Vector(-13, -4), Vector(13,-4),␣
↪Vector(13,4), Vector(-13,4), Vector(-13,-4)]),
                                  mask=HitboxMask.bullet, # tell physics engine about␣
↪object type
                                  collision_mask=HitboxMask.enemy, # tell physics␣
↪engine which objects it can collide with
                                  trigger_id=settings.COLLISION_TRIGGER_MG_BULLET #␣
↪ID to be used in custom collision handling function
                                  ))
```

Listing 43: objects/weapons/machine_gun.py

```
import registry
import settings
from objects.bullets.machine_gun_bullet import MachineGunBullet
from objects.weapons.base import WeaponBase
from kaa.geometry import Vector


class MachineGun(WeaponBase):

    def __init__(self):
        # node's properties
        super().__init__(sprite=registry.global_controllers.assets_controller.machine_
↪gun_img)

    def shoot_bullet(self):
        bullet_position = self.get_initial_bullet_position()
        bullet_velocity = Vector.from_angle_degrees(self.parent.rotation_degrees) *␣
↪settings.MACHINE_GUN_BULLET_SPEED
        self.scene.space.add_child(MachineGunBullet(position=bullet_position,␣
↪velocity=bullet_velocity,
                                                    rotation_degrees=self.parent.
↪rotation_degrees))
        # reset cooldown time
        self.cooldown_time_remaining =  self.get_cooldown_time()

    def get_cooldown_time(self):
        return 0.100
```

The above is very similar to the force gun. You may run the game and see how it looks. The main difference is that the machine gun bullets don't bounce back when colliding with enemies. In fact they're not affected at all by any collisions, and behave as if they had very large mass, pushing enemies with great energy. It's because they're kinematic bodies. We'll fix that in a moment, by writing our own collision handler.

1.5.8 Collisions handling

Let's implement a collision handler function to process collisions between machine gun bullet and enemy. This is where `trigger_id` values are being used. Put the following code in the `controllers/collisions_controller.py`:

Listing 44: controllers/collisions_controller.py

```
import settings

class CollisionsController:

    def __init__(self, scene):
        self.scene = scene
        self.space = self.scene.space
        self.space.set_collision_handler(settings.COLLISION_TRIGGER_MG_BULLET,␣
↪settings.COLLISION_TRIGGER_ENEMY,
                                         self.on_collision_mg_bullet_enemy)

    def on_collision_mg_bullet_enemy(self, arbiter, mg_bullet_pair, enemy_pair):
```

(continues on next page)

(continued from previous page)

```
        print("Detected a collision between MG bullet object {} hitbox {} and Enemy␣
↪object {} hitbox {}".format(
            mg_bullet_pair.body, mg_bullet_pair.hitbox, enemy_pair.body, enemy_pair.
↪hitbox))
```

The line where we call `set_collision_handler` on the scene's `SpaceNode` is where we tell the engine that we want our function to be called each time a collision between MG bullet and enemy occurs. We're using hitbox `trigger_id` here.

It is very important to realize that **a collision handler function can be called multiple times for given pair of colliding objects (even multiple times per frame)**. This can happen if object's hitboxes touch for the first time, then they either overlap or touch each other for some time and finally - they separate. The collision handler function will be called every frame, as long as the hitboxes touch or overlap. When they make apart, the collision handler function stops being called.

Collision handler function always has the three parameters:

- `arbiter` - arbiter object that includes additional information about collision. It has the following properties:
  - `space` - a `SpaceNode` where collision occurred.
  - `phase` - an enum value (`kaa.physics.CollisionPhase`), indicating collision phase. Available values are:
    - `kaa.physics.CollisionPhase.begin` - indicates that collision betwen two objects has started (their hitboxes have just touched or overlapped)
    - `kaa.physics.CollisionPhase.pre_solve` - indicates that two hitboxes are still in contact (touching or overlapping). It is called before the engine calculates the physics (e.g. velocities of both colliding objects)
    - `kaa.physics.CollisionPhase.post_solve` - like pre_solve, but called after the engine calculates the physics for the objects.
    - `kaa.physics.CollisionPhase.separate` - indicates that hitboxes of our two objects have separated - the collision has ended
- two "collision_pair" objects, corresponding with trigger_ids. Each collision pair object has two properties:
  - `body` - referencing `BodyNode` which collided
  - `hitbox` - referencing `HitboxNode` which collided (remember that body nodes can have multiple hitboxes - here we can know which of them has collided!)

Next, let's hook up the controller with the scene in `scenes/gameplay.py`'s `__init__`:

Listing 45: scenes/gameplay.py

```
from controllers.collisions_controller import CollisionsController

class GameplayScene(Scene):

    def __init__(self):
        # ......... rest of the function .........
        self.collisions_controller = CollisionsController(self)
```

Run the game and shoot the machine gun at enemies to see that collision handler function is called (the print message appears in your std out)

Now, let's implement enemies "staggering" when hit. Stagger will simply be a number of miliseconds when alternative frame is displayed.

Listing 46: objects/enemy.py

```
class Enemy(BodyNode):

    def __init__(self, position, hp=100, *args, **kwargs):
        # ......... rest of the function .......
        self.stagger_time_left = 0

    def stagger(self):
        # use the "stagger" sprite
        self.sprite = registry.global_controllers.assets_controller.enemy_stagger_img
        # stagger stops enemy from moving:
        self.velocity = Vector(0, 0)
        # track time for staying in the "staggered" state
        self.stagger_time_left = 150

    def recover_from_stagger(self):
        # start using the standard sprite animation again
        self.transition=NodeSpriteTransition(registry.global_controllers.assets_
↪controller.enemy_frames,
                                                    duration=max(200, random.
↪gauss(400, 100)), loops=0)

        self.stagger_time_left = 0
```

And track stagger time and recovery in the enemies controller:

Listing 47: controllers/enemies_controller.py

```
class EnemiesController:
    # ........ rest of the class ..........

    def update(self, dt):
        for enemy in self.enemies:
            # handle enemy stagger time and stagger recovery
            if enemy.stagger_time_left > 0:
                enemy.stagger_time_left -= dt
                if enemy.stagger_time_left <= 0:
                    enemy.recover_from_stagger()
```

Finally let's implement the proper collision handling logic when a machine gun bullet collides with an enemy. We would apply 10 HP damage and add a blood splatter animation at a place where collision occurred. If enemy HP drops below zero we remove the enemy from the scene and play enemy death animation.

Listing 48: controllers/collisions_controller.py

```
import math
import settings
import registry
import random
from kaa.physics import CollisionPhase
from kaa.nodes import Node
from kaa.geometry import Alignment

class CollisionsController:
    # ....... rest of the class ........
```

(continues on next page)

(continued from previous page)

```
    def on_collision_mg_bullet_enemy(self, arbiter, mg_bullet_pair, enemy_pair):
        print("Detected a collision between MG bullet object {} hitbox {} and Enemy␣
↪object {} hitbox {}".format(
            mg_bullet_pair.body, mg_bullet_pair.hitbox, enemy_pair.body, enemy_pair.
↪hitbox))

        if arbiter.phase == CollisionPhase.begin:
            enemy = enemy_pair.body
            enemy.hp -= 10
            # add the blood splatter animation to the scene
            self.scene.root.add_child(Node(z_index=900,
                                           transition=NodeSpriteTransition(
                                               registry.global_controllers.assets_
↪controller.blood_splatter_frames,
                                               duration=140),
                                           position=enemy.position, rotation=mg_
↪bullet_pair.body.rotation + math.pi,
                                           lifetime=140))
            # add a random bloodstain - make smaller ones more likely since it's a␣
↪small arms hit :)
            self.scene.root.add_child(Node(z_index=1, sprite=random.choices(
                registry.global_controllers.assets_controller.bloodstain_imgs,␣
↪weights=[5, 3, 1, 0.5])[0],
                                           position=enemy.position, rotation=mg_
↪bullet_pair.body.rotation + math.pi,
                                           lifetime=random.randint(20000, 40000)))
            if enemy.hp <= 0:
                # show death animation
                self.scene.root.add_child(Node(z_index=1,
                                               transition=NodeSpriteTransition(random.
↪choice(
                                                   registry.global_controllers.assets_
↪controller.enemy_death_frames),
                                                   duration=450),
                                               position=enemy.position,␣
↪rotation=enemy.rotation,
                                               origin_alignment=Alignment.right,
                                               lifetime=random.randint(10000, 20000)))
                # remove enemy node from the scene
                self.scene.enemies_controller.remove_enemy(enemy)
            else:
                enemy.stagger()

            mg_bullet_pair.body.delete()  # remove the bullet from the scene
            return 0 # tell the engine to ignore this collision
```

The bullet-enemy collision handling logic is rather self-explanatory but let's highlight a few things

First, note that we remove objects from the scene at the end of the function. Remember that when a `delete()` is called on an object we can no longer use its properties, even if we only want to read them!

Next, notice `return 0`. This tells the engine to ignore the collision effects. Normally, the bullet (kinematic body node) would push the enemy (dynamic body node), but we don't want this to happen - we just want the bullet to be destroyed on collision and we apply 'stagger'.

Run the game and enjoy shooting at enemies with machine gun, blood splatters and bodies falling down :)

1.5.9 Static BodyNodes

We won't add any static BodyNodes to the game, but they're the simplest form of nodes: they can collide with other objects but they themselves don't move. Use static BodyNodes when you're sure that an object won't transform in any way (move, scale or rotate). Using static BodyNodes instead of dynamic/kinematic BodyNodes with no velocity improves the performance.

1.5.10 Applying velocity to BodyNodes manually

Let's implement a simple AI for our enemies. Let's make each enemy be in one of the two modes:

- Moving to a waypoint - we'll pick a random point on the screen and enemy will move towards it, when it reaches it we'll randomize another point
- Moving towards player - enemy will simply move towards player's current position in a straight line

Let's define an enum:

Listing 49: common/enums.py

```
class EnemyMovementMode(enum.Enum):
    MoveToWaypoint = 1
    MoveToPlayer = 2
```

Then, let's add damping (a drag force working in entire space) to slow down enemies when they're moving freely due to collisions impulses (eg from Force gun bullet)

Listing 50: scenes/gameplay.py

```
# inside __init__:
self.space = SpaceNode(damping=0.3)
```

Next, let's modify the `Enemy` class:

Listing 51: objects/enemy.py

```
import random
from common.enums import EnemyMovementMode

class Enemy(BodyNode):

    def __init__(self, position, hp=100, *args, **kwargs):
        # ....... rest of the function  .......

        # 75% enemies will move towards player and 25% will move randomly
        if random.randint(0, 100) < 75:
            self.movement_mode = EnemyMovementMode.MoveToPlayer
        else:
            self.movement_mode = EnemyMovementMode.MoveToWaypoint
        self.current_waypoint = None  # for those which move to a waypoint, we'll␣
↪keep its corrdinates here
        self.randomize_new_waypoint()  # and randomize new waypoint

        self.acceleration_per_second = 300  # how fast will enemy accelerate
        self.max_velocity = random.randint(75, 125)  # we'll make enemy stop␣
↪accelerating if velocity is above this value
```

(continues on next page)

(continued from previous page)

```
    # ........ other methods ......

    def randomize_new_waypoint(self):
        self.current_waypoint = Vector(random.randint(50, settings.VIEWPORT_WIDTH-50),
                                       random.randint(50, settings.VIEWPORT_HEIGHT-
↪50))
```

Finally, let's implement the movement logic in the `EnemiesController` class

Listing 52: controllers/enemies_controller.py

```
from common.enums import EnemyMovementMode

class EnemiesController:
    # ....... rest of the class ....

    def update(self, dt):
        player_pos = self.scene.player_controller.player.position

        for enemy in self.enemies:
            # handle enemy stagger time and stagger recovery
            if enemy.stagger_time_left > 0:
                enemy.stagger_time_left -= dt
                if enemy.stagger_time_left <= 0:
                    enemy.recover_from_stagger()

            # handle enemy movement
            if enemy.movement_mode == EnemyMovementMode.MoveToWaypoint:
                # rotate towards the current waypoint:
                enemy.rotation_degrees = (enemy.current_waypoint - enemy.position).to_
↪angle_degrees()
                # # if we're less than 10 units from the waypoint, we randomize a new␣
↪one!
                if (enemy.current_waypoint - enemy.position).length() <= 10:
                    enemy.randomize_new_waypoint()
            elif enemy.movement_mode == EnemyMovementMode.MoveToPlayer:
                # rotate towards the player:
                enemy.rotation_degrees = (player_pos - enemy.position).to_angle_
↪degrees()
            else:
                raise Exception('Unknown enemy movement mode: {}'.format(enemy.
↪movement_mode))

            # if enemy velocity is lower than max velocity, then increment velocity.␣
↪Otherwise do nothing - the enemy
            # will be a freely moving object until the damping slows it down below␣
↪max speed
            if enemy.velocity.length() < enemy.max_velocity:
                # increment the velocity
                enemy.velocity += Vector.from_angle_degrees(enemy.rotation_degrees).
↪normalize()*\
                                  (enemy.acceleration_per_second*dt/1000)
```

Run the game and check it out. 75% of the enemies will walk towards the player while the other ones will wander randomly. What we're doing here is we accelerate enemies by incrementing their velocity every frame (as discussed before we're taking using dt in the formula to make it independent from the frame duration). We stop the velocity incremantation if enemy velocity exceeds the max value. When they're above max velocity they will behave as freely

moving objects and the drag force in the environment ("damping") will slow them down until they're below the max speed and start accelerating again.

An interesting effect of this model is inertia. Enemies can't change movement direction immediately where they stand, they need to decelerate and accelerate again. To lower the inertia you may increase the acceleration speed. For the freely moving enemies you may increase damping. Feel free to experiment with different values.

1.5.11 Applying impulses

Sometimes we don't want to apply velocity each frame. Instead we want to generate a single impulse that will affect object's velocity just once. A good example is the explosion that can push objects back. Let's illustrate this on the final weapon we'll have in the game: a grenade launcher. We want the grenade launcher to have the following features:

- Slow rate of fire (cooldown time of 1 second)
- Grenade exploding on collision with enemy, showing explosion animation
- Explosion dealing damage to all enemies in some radius, the further from the explosion center, the less damage dealt
- Explosion pushing all enemies in some radius, the further from the explosion center, the weaker the push back impulse
- We want pushing force to be a single-frame "impulse" applied to velocity, not some force applied each frame.

Let's get to it.

First, let's implement the grenade launcher bullet and grenade shooting logic. It is very similar to the machine gun logic, just using different sprite and a different hitbox shape for bullet, and a bigger cooldown time.

Listing 53: objects/bullets/grenade_launcher_bullet.py

```
import random
from kaa.physics import BodyNodeType, HitboxNode, BodyNode
from kaa.geometry import Circle
import registry
import settings
from common.enums import HitboxMask


class GrenadeLauncherBullet(BodyNode):

    def __init__(self, *args, **kwargs):
        super().__init__(sprite=registry.global_controllers.assets_controller.grenade_
↪launcher_bullet_img,
                         z_index=30,
                         body_type=BodyNodeType.kinematic,  # as we want to handle␣
↪collision effects on our own
                         lifetime=5000,  # will be removed from the scene␣
↪automatically after 5 secs
                         rotation_degrees=random.uniform(0, 360),  # a random␣
↪rotation between 0 and 360 degs
                         *args, **kwargs)
        self.add_child(HitboxNode(shape=Circle(radius=6),  # circular hitbox
              mask=HitboxMask.bullet,  # we are bullet
              collision_mask=HitboxMask.enemy,  # want to collide with objects whose␣
↪mask is enemy
              trigger_id=settings.COLLISION_TRIGGER_GRENADE_LAUNCHER_BULLET  # used␣
↪when registering collision handler function
```

(continues on next page)

(continued from previous page)

```
            ))
```

Listing 54: objects/weapons/grenade_launcher.py

```
import registry
import settings
import random
from objects.bullets.grenade_launcher_bullet import GrenadeLauncherBullet
from objects.weapons.base import WeaponBase
from kaa.geometry import Vector


class GrenadeLauncher(WeaponBase):

    def __init__(self):
        # node's properties
        super().__init__(sprite=registry.global_controllers.assets_controller.grenade_
↪launcher_img)

    def shoot_bullet(self):
        bullet_position = self.get_initial_bullet_position()
        bullet_velocity = Vector.from_angle_degrees(self.parent.rotation_degrees) *␣
↪settings.GRENADE_LAUNCHER_BULLET_SPEED
        self.scene.space.add_child(GrenadeLauncherBullet(position=bullet_position,␣
↪velocity=bullet_velocity))
        # reset cooldown time
        self.cooldown_time_remaining =  self.get_cooldown_time()

    def get_cooldown_time(self):
        return 1.0
```

Then, let's write a function that will apply explosion effects, such as dealing damage and pushing enemies back Here's where we reset enemy velocity thus generating an impulse which will push them back away from the explosion center.

Listing 55: controllers/enemies_controller.py

```
import random
import registry
import math
from common.enums import EnemyMovementMode
from objects.enemy import Enemy
from kaa.geometry import Vector, Alignment
from kaa.nodes import Node

class EnemiesController:

# ..... rest of the class ....

    def apply_explosion_effects(self, explosion_center, damage_at_center=100, blast_
↪radius=200,
                                pushback_force_at_center=500, pushback_radius=300):
        enemies_to_remove = []
        for enemy in self.enemies:
            # get the distance to the explosion
            distance_to_explosion = enemy.position.distance(explosion_center)
```

(continues on next page)

(continued from previous page)

```
            # if within pushback radius...
            if distance_to_explosion<=pushback_radius:
                # calculate pushback value, the further from the center, the smaller␣
↪it is
                pushback_force_val = pushback_force_at_center * (1 - (distance_to_
↪explosion/pushback_radius))
                # apply the pushback force by resetting enemy velocity
                enemy.velocity = (enemy.position-explosion_center).
↪normalize()*pushback_force_val

            # if within blast radius...
            if distance_to_explosion<=blast_radius:
                # calculate damage, the further from the center, the smaller it is
                damage = damage_at_center * (1 - (distance_to_explosion/blast_radius))
                # apply damage
                enemy.hp -= int(damage)
                # add the blood splatter animation over the enemy
                self.scene.root.add_child(Node(z_index=900,
                                               transition=NodeSpriteTransition(
                                                   registry.global_controllers.assets_
↪controller.blood_splatter_frames,
                                                   duration=140),
                                               position=enemy.position,␣
↪rotation=(enemy.position-explosion_center).to_angle() + math.pi,
                                               lifetime=140))

                if enemy.hp < 0:  # IZ DED!
                    # show the death animation (pick random sprite from few␣
↪animations we have loaded from one png file)
                    self.scene.root.add_child(Node(z_index=1,
                                                   ␣
↪transition=NodeSpriteTransition(random.choice(
                                                       registry.global_controllers.
↪assets_controller.enemy_death_frames),
                                                       duration=450),
                                                   position=enemy.position,␣
↪rotation=enemy.rotation,
                                                   origin_alignment=Alignment.right,
                                                   lifetime=random.randint(10000,
↪20000)))
                    # mark enemy for removal:
                    enemies_to_remove.append(enemy)

        # removed killed enemies
        for dead_enemy in enemies_to_remove:
            self.remove_enemy(dead_enemy)
```

Finally let's write a collision handler that will show explosion animation and call the `apply_explosion_effect` function we've just written.

Listing 56: controllers/collisions_controller.py

```
class CollisionsController:

    def __init__(self, scene):
```

(continues on next page)

(continued from previous page)

```
        # ....... rest of the function .........

        self.space.set_collision_handler(settings.COLLISION_TRIGGER_GRENADE_LAUNCHER_
↪BULLET, settings.COLLISION_TRIGGER_ENEMY,
                                         self.on_collision_grenade_enemy)

    # ...... rest of the class .......

    def on_collision_grenade_enemy(self, arbiter, grenade_pair, enemy_pair):

        if arbiter.phase == CollisionPhase.begin:
            # show explosion animation
            self.scene.root.add_child(Node(transition=NodeSpriteTransition(
                registry.global_controllers.assets_controller.explosion_frames,␣
↪duration=12*75),
                position=grenade_pair.body.position, z_index=1000, lifetime=12*75))
            # apply explosion effects to enemies (deal damage & push them back)
            self.scene.enemies_controller.apply_explosion_effects(grenade_pair.body.
↪position)

            grenade_pair.body.delete()  # remove the grenade from the scene
            return 0
```

Run the game, spawn a lot of enemies by pressing SPACE and have fun with the grenade launcher :) Be sure to verify they're being pushed back by the explosion and taking damage!

That concludes chapter 5. Let's *move on to chapter 6*, where we'll add some music and sound effects to our game.

1.6 Part 6: Sound effects and music

In this chapter we'll add sound effects and music to the game.

1.6.1 Loading sound effects from files

Loading sound effect from file is easy:

```
from kaa.audio import Sound
my_sound = Sound('/path/to/sound.wav')
```

Currently supported sound formats are:

- wav
- ogg

1.6.2 Playing sound effect

To play the sound effect:

```
my_sound.play(volume=0.9) # volume parameter ranging from 0 to 1, default is 1
```

You can play many sound effects simultaneously. There is a max limit of simultaneous sound that can be played. To change the limit use Scene's `audio.mixing_channels` property.

Note: Setting max limit to a very large number and playing very large number of sounds simultaneously may degrate performace of your game.

1.6.3 Stopping sound effect being played

When you call `play()` on a `Sound`, kaa will play the whole sound. If you want to stop playing the sound effect manually, you need to wait until next version of kaa because stopping sound effects is not yet implemented.

1.6.4 Loading music files from files

Loading music tracks is very similar to loading sound effects:

```
from kaa.audio import Music
my_music_track = Music('/path/to/music_track.wav')
```

Currently supported music formats are:

- wav
- ogg

1.6.5 Playing music track

To play the music track call `play` on your `Music` object:

```
my_music_track.play(volume=1.0)
```

You can play only one music track at a time. Playing new music track automatically stops the current track being played.

1.6.6 Stopping music track

If you want to just stop the current track being played without replacing it with a new track:

```
from kaa.audio import Music
Music.get_current().stop()
```

1.6.7 Knowing when music track has ended

Typically you will like to know when the current music track has ended so you can select a new one. To do this look for the audio events in the Scene's `events()` list:

```
class MyScene(Scene):

    def update(dt)

        for event in self.input.events():
            if event.audio: # check if it's an audio related event
                if event.audio.music_finished:
                    # do something when the track has finished playing ...
```

1.6.8 Full example

Let's use the sound and music in our tutorial game.

First, let's load all assets from files first, in our `AssetsController`

Listing 57: controllers/assets_controller.py

```
from kaa.audio import Sound, Music

class AssetsController:

    def __init__(self):

        # ..... rest of the function .....

        # Load all sounds
        self.mg_shot_sound = Sound(os.path.join('assets', 'sfx', 'mg-shot.wav'))
        self.force_gun_shot_sound = Sound(os.path.join('assets', 'sfx', 'force-gun-
↪shot.wav'))
        self.grenade_launcher_shot_sound = Sound(os.path.join('assets', 'sfx',
↪'grenade-launcher-shot.wav'))
        self.explosion_sound = Sound(os.path.join('assets', 'sfx', 'explosion.wav'))

        # Load all music tracks
        self.music_track_1 = Music(os.path.join('assets', 'music', 'track_1.wav'))
```

Let's play the music when the game starts.

Listing 58: main.py

```
with Engine(virtual_resolution=Vector(settings.VIEWPORT_WIDTH, settings.VIEWPORT_
↪HEIGHT)) as engine:
    # initialize global controllers and remember them in the registry
    registry.global_controllers.assets_controller = AssetsController()
    # play music
    registry.global_controllers.assets_controller.music_track_1.play()

    # .... rest of the code ....
```

Note: main.py isn't the best place to put this code. The music will stop playing after the track ends. To make it more usable maybe we should have a `MusicController` to manage tracks, and take care of starting new track when the previous ends? We'll leave this task to you :)

Let's play shooting sounds for the guns we have in the game:

Listing 59: objects/weapons/force_gun.py

```
class ForceGun(WeaponBase):

    def shoot_bullet(self):
        # .... rest of the function ....

        # play shooting sound
        registry.global_controllers.assets_controller.force_gun_shot_sound.play()
```

Listing 60: objects/weapons/grenade_launcher.py

```
class GrenadeLauncher(WeaponBase):

    def shoot_bullet(self):
        # .... rest of the function ....

        # play shooting sound
        registry.global_controllers.assets_controller.grenade_launcher_shot_sound.
↪play()
```

Listing 61: objects/weapons/machine_gun.py

```
class MachineGun(WeaponBase):

    def shoot_bullet(self):
        # .... rest of the function ....

        # play shooting sound
        registry.global_controllers.assets_controller.mg_shot_sound.play()
```

And the explosion sound:

Listing 62: controllers/enemies_controller.py

```
class EnemiesController:

    def apply_explosion_effects(self, explosion_center, damage_at_center=40, blast_
↪radius=150,
                                pushback_force_at_center=500, pushback_radius=300):
        # play explosion sound
        registry.global_controllers.assets_controller.explosion_sound.play()
        # .... rest of the function ....
```

Run the game and enjoy the experience with sounds and music. When you're ready, move on to the *part 7 of the tutorial* where we'll learn how to draw text.

1.7 Part 7: Drawing text

In this chapter we'll lear how to draw text in the game

1.7.1 Loading fonts from files

In order to draw a text, we must first load a font from a file. Like with images or sounds, it's very easy:

```
from kaa.fonts import Font
my_font = Font('/path/to/font.ttf')
```

Font formats currently supported by kaa:

- ttf

1.7.2 Drawing text

To draw a text, create a `TextNode` and add it to the scene. `TextNode` extends basic `Node` and therefore inherits all its properties - position, rotation, scale, color, origin_alignment etc. It adds the following new properties:

- `font` - A font to use when rendering text. Pass a `Font` instance.
- `text` - a string. A text you want to draw.
- `font_size` - an integer. Size of the text.
- `line_width` - an integer. Width of the text, in pixels. If set, the text will wrap automatically to fit this width. If not set, text won't wrap.
- `interline_spacing` - an integer. Space between lines of text in pixels. Used when the text wraps (due to line_width).
- `first_line_indent` - an integer. Indentation for the first line. Useful when you have multiple line texts (due to line_width)

1.7.3 Full example

Let's load a font from file and add some texts in the game.

Listing 63: controllers/assets_controller.py

```
from kaa.fonts import Font

class AssetsController:

    def __init__(self):
        # ... the rest of the function .....

        # Load all fonts
        self.font_1 = Font(os.path.join('assets', 'fonts', 'paladise-script.ttf'))
        self.font_2 = Font(os.path.join('assets', 'fonts', 'DejaVuSans.ttf'))
```

Listing 64: scenes/gameplay.py

```
import registry
import settings
from kaa.geometry import Vector, Alignment
from kaa.fonts import TextNode
from kaa.colors import Color

class GameplayScene(Scene):

    def __init__(self):
        super().__init__()
        self.frag_count = 0
        self.root.add_child(TextNode(font=registry.global_controllers.assets_
↪controller.font_1,
                                     origin_alignment=Alignment.left, position=Vector(10, 20),␣
↪font_size=40, z_index=1,
                                     text="WASD to move, mouse to rotate, left mouse button to␣
↪shoot"))
        self.root.add_child(TextNode(font=registry.global_controllers.assets_
↪controller.font_1,
```

(continues on next page)

(continued from previous page)

```
                                origin_alignment=Alignment.left, position=Vector(10, 45),␣
↪font_size=40, z_index=1,
                                text="1, 2, 3 - change weapons. SPACE - spawn enemy"))
        self.root.add_child(TextNode(font=registry.global_controllers.assets_
↪controller.font_2,
                                origin_alignment=Alignment.right, position=Vector(1910,␣
↪20), font_size=30, z_index=1,
                                color=Color(1, 0, 0, 1), text="Press Q to quit game"))
        self.frag_count_label = TextNode(font=registry.global_controllers.assets_
↪controller.font_1,
                                origin_alignment=Alignment.left, position=Vector(10, 70),␣
↪font_size=40, z_index=1,
                                color=Color(1, 1, 0, 1), text="")
        self.root.add_child(self.frag_count_label)
        # .... rest of the code

    def score_frag(self):
        # function for tracking frag count
        self.frag_count += 1
        self.frag_count_label.text = f"Frag Count: {self.frag_count}"
```

Run the game and check out the results!

Note: When adding `TextNode` to the scene it's important to give them proper `z_index`. Games will usually have some background image and you may often be wondering "why is that TextNode not visible"? Most likely it's because of `z_index` being too low and some other sprite is rendering in front of it!

1.7.4 Updating text

Updating text property of the `TextNode` is a performance-heavy operation and you should avoid updating `text` property on each frame (unless it's really needed). In our case, we only need to update the frag count when an enemy is killed. We've already written a `score_frag` function, let's now call it:

Listing 65: controllers/enemies_controller.py

```
class EnemiesController:

    def remove_enemy(self, enemy):
        self.enemies.remove(enemy)  # remove from the internal list
        enemy.delete()  # remove from the scene
        # increment the frag counter
        self.scene.score_frag()
```

1.7.5 Transforming text

Since text nodes are regular Nodes, you can use all of base `Node` properties to transform them, e.g. reposition, rotate, scale, etc.

```
my_text_node.rotation_degrees = 45
my_text_node.scale = Vector(0.5, 0.75)
```

Text Nodes can also be a child nodes of other nodes, and can have child nodes themselves.

```
tn = TextNode(font = my_font, text="Hello world")
tn.add_child(Node(sprite=my_sprite))
```

Let's move on, *to the next part of the tutorial*

1.8 Part 8: Working with multiple scenes

So far we have had just one Scene in our game, the `GameplayScene`. Let's add two more: for the title screen and for the pause screen. Even though we'll have 3 scenes created in the game, only one of them can be active at a time. It means that only active scene will render its nodes on the screen, run the `update()` loop and receive input events. The other scenes will become "freezed" until one of them is activated again. Their `update()` loops won't be called, no input events will be published to them, no nodes present in those scenes will be drawn on the screen etc.

1.8.1 How to activate a new scene

To make another scene active, get an engine object first, and then call `change_scene(new_scene)` method.

To get an engine:

```
from kaa.engine import get_engine
engine = get_engine()
engine.change_scene(some_new_scene)
```

Each scene has the engine object stored under `self.engine` so you can get it from there as well:

```
# .... inside kaa.engine.Scene class method ....
self.engine.change_scene(some_new_scene)
```

1.8.2 How to create a new scene

Let's write two more scenes:

- `GameTitleScene` - Will be activated when the game starts. The scene will be a welcome screen, showing a logo and allowing to start the game or exit it.
- `PauseScene` - Will be activated when pressing ESC during gameplay. Will show a simple screen allowing to abort game (return to title screen) or resume game (return to gameplay scene)

Listing 66: scenes/title_screen.py

```
import registry
import settings
from kaa.engine import Scene
from kaa.input import Keycode, MouseButton
from kaa.nodes import Node
from kaa.geometry import Vector, Alignment
from kaa.fonts import TextNode


class TitleScreenScene(Scene):

    def __init__(self):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
        self.root.add_child(Node(sprite=registry.global_controllers.assets_controller.
↪title_screen_background_img,
                                 z_index=0, position=Vector(0,0), origin_
↪alignment=Alignment.top_left))
        self.root.add_child(TextNode(font=registry.global_controllers.assets_
↪controller.font_2, font_size=30,
                                     position=Vector(settings.VIEWPORT_WIDTH/2, 500),␣
↪text="Click to start the game",
                                     z_index=1, origin_alignment=Alignment.center))
        self.root.add_child(TextNode(font=registry.global_controllers.assets_
↪controller.font_2, font_size=30,
                                     position=Vector(settings.VIEWPORT_WIDTH/2, 550),␣
↪text="Press ESC to exit",
                                     z_index=1, origin_alignment=Alignment.center))

    def update(self, dt):

        for event in self.input.events():

            if event.keyboard_key:
                if event.keyboard_key.is_key_down and event.keyboard_key.key ==␣
↪Keycode.escape:
                    self.engine.quit()

            if event.mouse_button:
                if event.mouse_button.is_button_down and event.mouse_button.button ==␣
↪MouseButton.left:
                    self.engine.change_scene(registry.scenes.gameplay_scene)
```

Nothing unusual here, just the stuff we already know: the scene is pretty static, with just a background image and two labels. Mouse click changes the scene to gameplay and ESC quits the game. It won't work yet, because registry object does not store gameplay_scene yet, but we'll get there.

For now, let's add the pause scene. It is very similar to the title screen scene:

Listing 67: scenes/pause.py

```
import registry
import settings
from kaa.engine import Scene
from kaa.input import Keycode
from kaa.geometry import Vector, Alignment
from kaa.fonts import TextNode


class PauseScene(Scene):

    def __init__(self):
        super().__init__()
        self.root.add_child(TextNode(font=registry.global_controllers.assets_
↪controller.font_2, font_size=40,
                                     position=Vector(settings.VIEWPORT_WIDTH/2, 300),␣
↪text="GAME PAUSED",
                                     z_index=1, origin_alignment=Alignment.center))
        self.root.add_child(TextNode(font=registry.global_controllers.assets_
↪controller.font_2, font_size=30,
                                     position=Vector(settings.VIEWPORT_WIDTH/2, 550),␣
↪text="Press ESC to resume",
```

(continues on next page)

(continued from previous page)

```
                                    z_index=1, origin_alignment=Alignment.center))
        self.root.add_child(TextNode(font=registry.global_controllers.assets_
↪controller.font_2, font_size=30,
                                    position=Vector(settings.VIEWPORT_WIDTH/2, 650),␣
↪text="Press q to abort",
                                    z_index=1, origin_alignment=Alignment.center))


    def update(self, dt):
        for event in self.input.events():
            if event.keyboard_key and event.keyboard_key.is_key_down:
                if event.keyboard_key.key == Keycode.escape:
                    self.engine.change_scene(registry.scenes.gameplay_scene)
                if event.keyboard_key.key == Keycode.q:
                    self.engine.change_scene(registry.scenes.title_screen_scene)
```

Let's now make a small modification to the `GameplayScene` allowing to change scene to pause, when player presses ESC.

Listing 68: scenes/gameplay.py

```
def update(self, dt):
    # .... other code ....

    for event in self.input.events():
        # .... other code ....
        if event.keyboard_key and event.keyboard_key.is_key_down:
            if event.keyboard_key.key == Keycode.escape:
                self.engine.change_scene(registry.scenes.pause_scene)
```

Finally, let's create all our scenes in the `main.py` and add them to the registry to make the `change_scene` calls work!

Listing 69: main.py

```
from scenes.pause import PauseScene
from scenes.title_screen import TitleScreenScene

with Engine(virtual_resolution=Vector(settings.VIEWPORT_WIDTH, settings.VIEWPORT_
↪HEIGHT)) as engine:
    # .... rest of the function ....

    # initialize scenes and remember them in the registry
    registry.scenes.gameplay_scene = GameplayScene()
    registry.scenes.title_screen_scene = TitleScreenScene()
    registry.scenes.pause_scene = PauseScene()
    engine.run(registry.scenes.title_screen_scene)
```

Run the game. Isn't it much better with all those different screens? I think it is!

1.8.3 Starting a new game

If you test the flow of the game, you'll notice the following bug: aborting game and then starting new game just returns to the previous state of the scene: all monsters are where they were left, frag count is not reset and so on. It's because `change_scene` does not destroy scene state it just runs a new scene and freezes all other scenes, as we stated earlier.

A bug needs fixing! Let's refactor the `TitleScreenScene` a little bit:

Listing 70: scenes/title_screen.py

```
from scenes.gameplay import GameplayScene

class TitleScreenScene(Scene):
    # .... rest of the class ....

    def start_new_game(self):
        registry.scenes.gameplay_scene = GameplayScene()
        self.engine.change_scene(registry.scenes.gameplay_scene)

    def update(self, dt):
        for event in self.input.events():
            # ... other code ...
            if event.mouse_button and event.mouse_button.is_button_down and event.
↪mouse_button.button == MouseButton.left:
                self.start_new_game()
```

We simply create the new instance of GameplayScene before telling engine to change to that scene. Run the game again and enjoy the full experience of multiple scenes :)

1.8.4 Scene's on_enter and on_exit methods

Scene has two methods `on_enter` and `on_exit`. They're being used when you call `change_scene` so you can do some additional initialization or cleanup before the scene loads.

```
class Gameplay(Scene):

    def on_enter(self):
        # do something when active scene changes TO this scene.

    def on_exit(self):
        # do something when active scene changes FROM this scene.
```

Let's move on to *the next part of the tutorial* where we'll learn few things about the camera.

1.9 Part 9: The camera

Camera projects the scene into your 2D display. Controlling the camera allows us to add few extra visual effects.

1.9.1 Getting the camera

Camera is available directly in the scene:

```
class SomeScene(kaa.engine.Scene):

    def foo(self):
        hi_i_am_camera = self.camera # the camera is here!
```

1.9.2 Camera properties

The camera object has the following properties:

- `position` - allows to move the camera
- `rotation` - allows to rotate the camera (using radians)
- `rotation_degrees` - allows to rotate the camera (using degrees)
- `scale` - allows for applying a zoom in / zoom out effect

The camera object has also the following method:

- `unproject_position(position_vector)` - a helper function that transforms a current screen position into absolute position by applying current camera transformations. Practical use is illustrated below.

1.9.3 Full Example

Let's use the camera in our game:

- arrow keys to move the camera up, down, left right
- page up and page down keys to change the camera's scale up and down
- home and end keys to rotate the camera clockwise and anti-clockwise

Let's add the following code to the `GameplayScene`

Listing 71: scenes/gameplay.py

```
class GameplayScene(Scene):
    # ... rest of the class ...

    def update(self, dt):
        # ... other code ....

        if self.input.keyboard.is_pressed(Keycode.left):
            self.camera.position -= Vector(-0.1 * dt, 0)
        if self.input.keyboard.is_pressed(Keycode.right):
            self.camera.position -= Vector(0.1 * dt, 0)
        if self.input.keyboard.is_pressed(Keycode.up):
            self.camera.position -= Vector(0, -0.1 * dt)
        if self.input.keyboard.is_pressed(Keycode.down):
            self.camera.position -= Vector(0, 0.1 * dt)

        if self.input.keyboard.is_pressed(Keycode.pageup):
            self.camera.scale -= Vector(0.001*dt, 0.001*dt)
        if self.input.keyboard.is_pressed(Keycode.pagedown):
            self.camera.scale += Vector(0.001*dt, 0.001*dt)

        if self.input.keyboard.is_pressed(Keycode.home):
            self.camera.rotation_degrees += 0.03 * dt
        if self.input.keyboard.is_pressed(Keycode.end):
            self.camera.rotation_degrees -= 0.03 * dt
```

Run the game and see how you can control the camera in the gameplay scene using arrow keys, page up/down and home/end keys.

Have you noticed? When you transform the camera (especially when you rotate it) and then shoot your guns, the bullets don't fly where they should... If the mouse pointer is in the (0,0) position i.e. top-left of the screen, the bullets

don't fly to that exact place but to the top-left corner **of the projected image of the scene**. It's not a bug, it's a feature! Point (0,0) of the scene always is a (0,0) regardless of transformations applied to the camera!

In other words, if we apply a transformation to the camera we also need to apply the same transformation to the `get_mouse_position()` function! That's where camera's `unproject_position(position_vector)` function can help.

Let's modify the code in `PlayerController` where `get_mouse_position()` is used.

Listing 72: controllers/player_controller.py

```
# that fragment inside update() function....
elif event.keyboard_key.key == Keycode.space:
    self.scene.enemies_controller.add_enemy(Enemy(position=self.scene.camera.
↪unproject_position(
        self.scene.input.mouse.get_position()), rotation_degrees=random.randint(0,
↪360)))

# another fragment inside update() function:
mouse_pos = self.scene.camera.unproject_position(self.scene.input.mouse.get_
↪position())
```

Run the game again and verify that shooting guns and spawning enemies have been fixed.

Moving the player is more interesting problem, but we won't change it now. After all, the player always moves the same way it's just the way we look at it that changes!

1.9.4 There isn't a "global" camera, each scene has its own

Each scene has its own camera, so if you apply transformation to a camera in scene A, and then change the scene to B then the camera in scene B will not be affected by those transformations!

That's all you need to know about camera for now. Let's move on to the *next part of the tutorial*.

1.10 Part 10: Transitions

We're already familiar with the `SpriteNodeTransition` object which we used to make Node's sprite change, creating a frame-by-frame animation effect. We mentioned that transitions are much general and powerful mechanism. It's time to explain what they are and how to use them.

When writing a game you'll often want to apply a set of known transformations to an object. For example, you want your object to move 100 pixels to the right, then wait 3 seconds and return 100 pixels to the left. Or you want to implement pulsation effect where an object would smoothly change its scale between some min and max values. There's an unlimited number of such visual transformations that you may want in your games as they greatly improve the game experience.

You can of course implement all this by having a set of boolean flags, time trackers, etc. and use all those helper variables to change the desired properties of your nodes over time manually. But there is an easier way: Transitions.

A single Transition object is a 'recipe' of how a given property of a Node (position, scale, rotation, etc.) should change over time. Transition can be applied to object once, given number of times or in a loop. You can chains transitions to run serially or in parallel.

It's best to illustrate on an example, so let's do it!

1.10.1 Adding a Transition to a Node

Let's practice transitions on a text node we have in the title screen.

Let's start by refactoring the code in `__init__`:

Listing 73: scenes/title_screen.py

```
class TitleScreenScene(Scene):

    def __init__(self):
        # ... cut the rest of the function ....
        self.exit_label = TextNode(font=registry.global_controllers.assets_controller.
↪font_2, font_size=30,
                                   position=Vector(settings.VIEWPORT_WIDTH/2, 550),␣
↪text="Press ESC to exit",
                                   z_index=1, origin_alignment=Alignment.center)
        self.root.add_child(self.exit_label)
        self.transitions_fun_stuff()
```

Then add the `transitions_fun_stuff` method:

Listing 74: scenes/title_screen.py

```
from kaa.transitions import *

def transitions_fun_stuff(self):
    my_transition = NodePositionTransition(Vector(300, 850), duration=3000)
    self.exit_label.transition = my_transition
```

Run the game and see the label moving from its original position to (300, 850), the movement takes 3 seconds! We did not have to change its position manually inside `update()`, it all happened automatically. Isn't it cool?

1.10.2 Changing a value incrementally

The transition we wrote takes node position and changes it (over 3 seconds) to the final value. But what if we don't want to move a node to a know position, but just 50 pixels to the left and 200 pixels down?

Listing 75: scenes/title_screen.py

```
def transitions_fun_stuff(self):
    my_transition = NodePositionTransition(Vector(-50, 200), duration=3000,
                                           advance_method=AttributeTransitionMethod.
↪add)
    self.exit_label.transition = my_transition
```

Available `advance_method` values are:

- kaa.transitions.AttributeTransitionMethod.set - the default mode. The target value is set directly.
- kaa.transitions.AttributeTransitionMethod.add - The target value will be calculated by adding operation
- kaa.transitions.AttributeTransitionMethod.multiply - The target value will be calculated by multiplying operation

1.10.3 Running transition back and forth

To run transition back and forth simply set `back_and_forth=True` on a transition object:

Listing 76: scenes/title_screen.py

```
def transitions_fun_stuff(self):
    my_transition = NodePositionTransition(Vector(-50, 200), duration=3000,
                                           advance_method=AttributeTransitionMethod.
↪add,
                                           back_and_forth=True)
    self.exit_label.transition = my_transition
```

Notice that the 3000 milisecond is the one-way time duration. Total transition duration back and forth takes 6000 miliseconds

1.10.4 Running transition specific number of times

To run transition specific number of times, set `loops` on a transition object to a desired value:

Listing 77: scenes/title_screen.py

```
def transitions_fun_stuff(self):
    my_transition = NodePositionTransition(Vector(-50, 200), duration=3000,
                                           advance_method=AttributeTransitionMethod.
↪add,
                                           back_and_forth=True, loops=3)
    self.exit_label.transition = my_transition
```

Run it and see that it moves back and forth 3 times.

Note: See what happens if you set loops to some value without `back_and_forth` set to `False`

1.10.5 Running transition infinite number of times

To run transition in an infinite loop set `loops` on a transition object to 0.

1.10.6 All types of transitions

We've learned about `NodePositionTransition` but what other transitions are available?

- `kaa.transitions.NodeSpriteTransition` - changes sprite of a node (we've already learned that)
- `kaa.transitions.NodePositionTransition` - changes position of a node
- `kaa.transitions.NodeRotationTransition` - changes rotation of a node
- `kaa.transitions.NodeScaleTransition` - changes scale of a node
- `kaa.transitions.NodeColorTransition` - changes color of a node
- `kaa.transitions.BodyNodeVelocityTransition` - changes velocity of a node (applicable to BodyNodes only)

- `kaa.transitions.BodyNodeAngularVelocityTransition` - changes angular velocity of a node (applicable to BodyNodes only)
- `kaa.transitions.NodeTransitionDelay` - waits for given number of miliseconds - useful when you chain few transitions together

It is also possible to write custom transitions, it's covered further below.

1.10.7 Chaining transitions

Let's build a chain of transitions: first we want the node to change its position, then rotate, then wait 0.5 second, then scale, and finally change color. To build such a sequence we'll use `NodeTransitionsSequence`

Listing 78: scenes/title_screen.py

```
from kaa.colors import Color
import math

def transitions_fun_stuff(self):
    move_transition = NodePositionTransition(Vector(-50, 200), duration=1000, advance_
↪method=AttributeTransitionMethod.add)
    rotate_transition = NodeRotationTransition(2*math.pi, duration=1000) # rotate 180␣
↪degrees (2*pi radians)
    wait_transition = NodeTransitionDelay(duration=500)
    scale_transition = NodeScaleTransition(Vector(2, 2), duration=1000) # enlarge␣
↪twice
    color_transition = NodeColorTransition(Color(1, 0, 0, 1), duration=1000) # change␣
↪color to red
    transition_sequence = NodeTransitionsSequence([move_transition, rotate_transition,
↪ wait_transition,
                                                   scale_transition, color_
↪transition])
    self.exit_label.transition = transition_sequence
```

Run the game and enjoy the nice transition sequence!

`NodeTransitionsSequence` has two already known properties: `back_and_forth` and `loops`. You can use them to run **the whole sequence** back and forth, specific number of times or in an infinite loop.

1.10.8 Knowing that a transition has ended

Sometimes we may want to be able to run some code when transition has ended, or when we reached some point in a chain of transition. We can use `NodeTransitionCallback`. It's only parameter is a callable. Let's show this on an example:

Listing 79: scenes/title_screen.py

```
def transition_callback_function(self, node):
    # play explosion sound
    registry.global_controllers.assets_controller.explosion_sound.play()

def transitions_fun_stuff(self):
    move_transition = NodePositionTransition(Vector(-50, 200), duration=1000, advance_
↪method=AttributeTransitionMethod.add)
    callback_transition = NodeTransitionCallback(self.transition_callback_function) #␣
↪call that function
```

(continues on next page)

(continued from previous page)

```
    rotate_transition = NodeRotationTransition(2*math.pi, duration=1000) # rotate 180␣
↪degrees (2*pi radians)
    wait_transition = NodeTransitionDelay(duration=500)
    scale_transition = NodeScaleTransition(Vector(2, 2), duration=1000) # enlarge␣
↪twice
    color_transition = NodeColorTransition(Color(1, 0, 0, 1), duration=1000) # change␣
↪color to red
    transition_sequence = NodeTransitionsSequence([move_transition, callback_
↪transition,
                                                   rotate_transition, wait_transition,
                                                   scale_transition, color_
↪transition])
    self.exit_label.transition = transition_sequence
```

It's pretty self-explanatory isn't it? callback_transition is executed between move_transition and rotate_transition therefore we hear explosion sound at that very moment.

1.10.9 Running transitions in paralel

Let's say we want to run some transitions (or sequences of those) in paralel. It's quite easy: we need to use `NodeTransitionsParallel`. Let's have our node rotate, scale, change color and move at the same time.

Listing 80: scenes/title_screen.py

```
def transitions_fun_stuff(self):
    rotate_transition = NodeRotationTransition(2*math.pi, duration=1000) # rotate 180␣
↪degrees (2*pi radians)
    scale_transition = NodeScaleTransition(Vector(2, 2), duration=1000) # enlarge␣
↪twice
    color_transition = NodeColorTransition(Color(1, 0, 0, 1), duration=1000) # change␣
↪color to red

    move_transition1 = NodePositionTransition(Vector(-200, 0), duration=1000,
                                              advance_method=AttributeTransitionMethod.add)
    move_transition2 = NodePositionTransition(Vector(200, 200), duration=1000,
                                              advance_method=AttributeTransitionMethod.add)
    move_transition3 = NodePositionTransition(Vector(200, -200), duration=1000,
                                              advance_method=AttributeTransitionMethod.add)
    move_transition4 = NodePositionTransition(Vector(-200, 0), duration=1000,
                                              advance_method=AttributeTransitionMethod.add)

    move_sequence = NodeTransitionsSequence([move_transition1, move_transition2, move_
↪transition3, move_transition4], loops=0)
    paralel_sequence = NodeTransitionsParallel([rotate_transition, scale_transition,␣
↪color_transition], back_and_forth=True, loops=0)

    # run both the movement sequence and rotate+scale+color sequence in paralel
    self.exit_label.transition = NodeTransitionsParallel([
        move_sequence, paralel_sequence])
```

Note that `NodeTransitionsParallel` has two already known properties: `back_and_forth` and `loops`.

You can nest transition sequences in other sequences, run such nested sequences in paralel and so on. Be aware on which level you set the `back_and_forth` and `loops` param values. Feel free to experiment with transitions on your own.

1.10.10 Contradictory transitions?

What happens if you try to run two position transitions in paralel: one moving a node 100 pixels to the right and the other moving it 100 pixels to the left. Contrary to intuition, they won't cancel out (regardless of `advance_method` being add or set). If there are two or more transitions of the same type running in paralel, then the one which is later in the list will be used and the preceding ones will be ignored.

1.10.11 Implementing custom transitions

You can implement your own transition, where you can fully control what's happening with the node over time.

Use `CustomNodeTransition` class. It takes 3 parameters:

- A callable with one parameter of type `<Node>`. This function will be called once, when the transition is assigned to a Node (it will pass that Node as parameter). Imeplement this function to return a state.
- A callable with three parameters: `state`, `node` and `t`. It will be called every frame during which the transition is in effect. State parameter is an object you prepared in the previous callable. Node parameter is the node that's transitioning. t is a value between 0 and 1 indicating time progress of present transition cycle
- A numerical value - duration of transition in miliseconds

`CustomNodeTransition` also has the `back_and_forth` and `loops` described in sections above.

1.10.12 Different easing patterns

As you probably noticed, transitions change the property of a node over time in a linear fashion. In other words, if transition orders the node to change rotation by 100 degrees in 10 seconds then the node will progress at a steady rate of 10 degrees per second.

Future kaa versions will have more types of "easing functions", other than linear, expect something similar to this

Let's move on to *the last part of the tutorial* where we'll build the game as executable file (.exe on Windows or binary executable on Linux)

1.11 Part 11: Building executable file and distributing via Steam

When distributing your game to other people (via Steam or other platform), you cannot give them a bunch of .py files. A distributable package must include a "native" executable file (exe on Windows or binary executable on Linux).

There are few tools that build native executables from python scripts. We'll use pyinstaller.

First, you need to install pyinstaller.

```
pip install pyinstaller
```

Then, navigate to the foder with main.py and run the following command:

```
pyinstaller --onefile --windowed --hidden-import numbers --icon assets\gfx\icon.ico␣
↪main.py
```

If the command ran successfully, pyinstaller should create the following folders/files in your project folder:

- dist folder - this is where you'll find the game executable ("main.exe" on Windows or "main" binary executable on Linux)
- build folder - just pyinstaller's build stuff

- main.spec file

To complete the work, copy the assets folder to the dist folder and try running the executable file.

Did it work? Congratulations, you have completed the tutorial and wrote a fully functional game! Don't hesitate to show it to your friends and family :)

Few remarks on switches we used in the `pyinstaller` command:

- `icon` - adds an icon to the exe file, to replace the ugly default icon
- `hidden-import numbers` - tells pyinstaller to import an additional dependency used by kaa which is not exposed directly, thus invisible to pyinstaller
- `onefile` - tells pyinstaller to add all dependencies to form just one executable file. Without this flag, dist folder will have a bunch of other lib files which you'll need to distribute with the game.
- `windowed` - tells pyinstaller that it's not a python script but a windowed app

Check out the pyinstaller documentation for much better description of all available options and their meaning.

1.11.1 Troubleshooting

- The executable was built successfully but fails to run, showing just "failed to execute script 'main'"? Delete the dist and build folders and run the pyinstaller command again, **without :code:'–onefile' and :code:'–windowed' options**. Then run the game **from the command line** (cmd.exe on Windows or terminal on Linux). It will print out python stack trace which hopefully will tell you more about the problem.
- if `pyinstaller` command did not complete successfully, check out the error message and look at the logs (inside "build" folder which will also.

1.11.2 Distributing kaa games on steam

Once you have distributable package (assets + binary executable) you can distribute it via Steam. Whe you configure your game for distribution in the Steamworks panel, be sure to go to Installation->Redistributable Packages and select "Visual C++ Redist 2017" and "DirectX June 2010"

1.11.3 Games made with kaa

"Git Gud or Get Rekt!" - retro space shooter, available for free on Steam

Did you make your own game with the kaa engine? Let us know! We'll be more than happy to include it on the list.

CHAPTER

TWO

KAA ENGINE REFERENCE

2.1 `audio` — Sound effects and music

2.1.1 `Sound` reference

class audio.**Sound**(*sound_filepath*, *volume=1.0*)

A *Sound* object represents a sound effect. Multiple sound effects can be played simultaneously.

sound_filepath argument must point to a sound file in a compatible format. Currently supported formats are:

- wav
- ogg

volume paramter must be a value between 0 and 1

Instance properties:

Sound.**volume**

Gets or sets a default volume of the sound effect.

Instance methods:

Sound.**play**(*volume=1.0*)

Plays the sound effect.

Volume is a value between 0 and 1. The volume is modified by the master sound volume level setting.

Refer to *engine.AudioManager* documentation on how to set the master volume for sounds.

Multiple sound effects can be played simultaneously, up to a limit set on the *AudioManager.mixing_channels* property.

The play() method is a simple "fire and forget" mechanism. It does not allow you to stop, pause or resume the sound. If you need more control on how the sound effects playback, use the *SoundPlayback* wrapper.

2.1.2 `SoundPlayback` reference

class audio.**SoundPlayback**(*sound*, *volume=1.0*)

A wrapper class for *Sound* objects, offering more control over sound effects playback.

The `sound` parameter must be a *Sound* instance.

Volume must be a value between 0 and 1.

Instance properties:

SoundPlayback.**sound**
Read only. Returns the wrapped *Sound* instance

SoundPlayback.**status**
Read only. Returns the sound status, as *AudioStatus* enum value.

SoundPlayback.**is_playing**
Read only. Returns `True` if the sound is playing.

SoundPlayback.**is_paused**
Read only. Returns `True` if the sound is paused.

SoundPlayback.**volume**
Gets or sets the volume. Must be a number between 0 and 1.

Instance methods:

SoundPlayback.**play**(*loops=1*)
Plays the sound effect.

The `loops` parameter is how many times the sound should play. Set to 0 to play the sound in the infinite loop.

Multiple sound effects can be played simultaneously, up to a limit set on the *AudioManager.mixing_channels* property.

Use *stop()*, *pause()* and *resume()* methods to control the sound playback.

SoundPlayback.**stop**()
Stops the sound playback if it's playing or paused.

SoundPlayback.**pause**()
Pauses the sound playback if it's playing.

SoundPlayback.**resume**()
Resumes the sound playback if it's paused.

2.1.3 `Music` reference

class audio.**Music**(*music_filepath*, *volume=1.0*)
A *Music* object represents a single music track. There's more control over playing Music tracks than Sounds as you can pause, resume or stop them on demand. Only one music track can be played at a time.

music_filepath argument must point to a soundtrack file in a compatible format. Currently supported formats are:

- wav
- ogg

Class methods

classmethod Music.**get_current**()
Returns *Music* instance currently being played

Instance properties

Music.**status**
Read only. Returns the status of the Music track, as *AudioStatus* enum value.

Music.**is_playing**
Read only. Returns `True` if the music is playing.

Music.**is_paused**
Read only. Returns `True` if the music is paused.

`Music.`**`volume`**
Gets or sets a default volume of the music track.

Instance methods

`Music.`**`play`**(*volume=1.0*)
Starts playing the music track. If another music track is playing it is automatically stopped.

Volume is a value between 0 and 1. The volume is modified by the master music volume level setting.

Refer to *`engine.AudioManager`* documentation on how to set the master volume for music.

`Music.`**`pause`**()
Pauses the music track currently being played. Can be resumed with *`Music.resume()`* method

`Music.`**`resume`**()
Resumes music track paused by *`Music.pause()`*. If the track is not paused, it does nothing.

`Music.`**`stop`**()
Stops the music track.

2.1.4 `AudioStatus` reference

`class` `audio.`**`AudioStatus`**
Enum type used for referencing sound or music status when working with *`Music`*, *`Sound`* and *`SoundPlayback`* objects. It has the following values:

- `AudioStatus.playing`
- `AudioStatus.paused`
- `AudioStatus.stopped`

2.2 `colors` — Wrapper class for colors

2.2.1 `Color` reference

Constructor:

`class` `colors.`**`Color`**(*r=0.0*, *g=0.0*, *b=0.0*, *a=1.0*)
A *`Color`* represents a color in RGBA format. Color is a property attribute of a *`nodes.Node`* instance and all its subclasses e.g. *`physics.BodyNode`*, *`physics.BodyNode`*, *`fonts.TextNode`* etc.

Giving *`nodes.Node`* a color tints this node's `geometry.Shape` in that color. In case of text nodes it sets the color of the text.

Parameters r, g, b and a are red, green, blue and alpha. They take values between 0 and 1.

Instance properties (read only):

`Color.`**`r`**
Returns red value

`Color.`**`g`**
Returns green value

`Color.`**`b`**
Returns blue value

`Color.a`
Returns blue value

Class methods:

classmethod `Color.from_int`(*r=0*, *g=0*, *b=0*, *a=0*)
Allows to construct a *Color* instance from integer parameters: r, g, b and a must be integers between 0 and 255

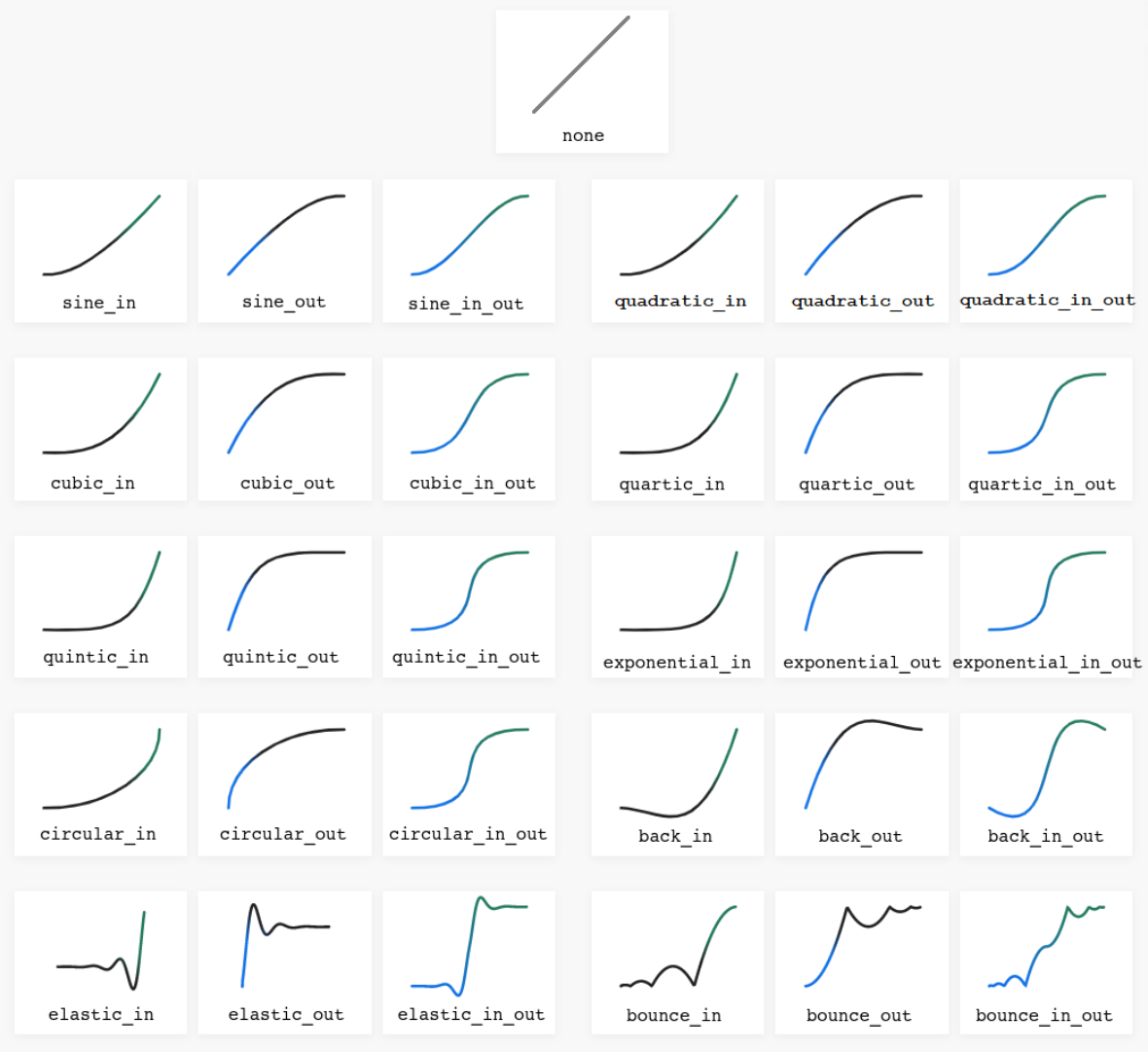
2.3 easings — Easing effects for transitions

2.3.1 Easing reference

class easings.**Easing**
Enum type used for referencing easing types to work with transitions. *Read more about Transitions here*. It has the following values:

- `Easing.none` - default easing, representing a linear progress
- `Easing.back_in`
- `Easing.back_in_out`
- `Easing.back_out`
- `Easing.bounce_in`
- `Easing.bounce_in_out`
- `Easing.bounce_out`
- `Easing.circular_in`
- `Easing.circular_in_out`
- `Easing.circular_out`
- `Easing.cubic_in`
- `Easing.cubic_in_out`
- `Easing.cubic_out`
- `Easing.elastic_in`
- `Easing.elastic_in_out`
- `Easing.elastic_out`
- `Easing.exponential_in`
- `Easing.exponential_in_out`
- `Easing.exponential_out`
- `Easing.quadratic_in`
- `Easing.quadratic_in_out`
- `Easing.quadratic_out`
- `Easing.quartic_in`
- `Easing.quartic_in_out`

- Easing.quartic_out
- Easing.quintic_in
- Easing.quintic_in_out
- Easing.quintic_out
- Easing.sine_in
- Easing.sine_in_out
- Easing.sine_out



2.3.2 `ease()` reference

easings.**ease**(*easing*, *t*)

Calculates the rate of change at time t for specific easing. The `t` parameter should be a float with a value between 0 (start of transition) and 1 (end of transition). The `easing` must be an *easings.Easing* value.

Returned value is a float.

```
print("Half into transition time, the rate value with the default easing is {}".
↪format(ease(Easing.none, 0.5)))
print("Half into transition time, the rate value with exponential easing is {}".
↪format(ease(Easing.exponential_in, 0.5)))
```

2.3.3 `ease_between()` reference

easings.**ease_between**(*easing*, *t*, *a*, *b*)

Calculates the actual value transitioning from `a` to `b` at time `t` using given `easing`.

The a and b parameters must be either floats of vectors (*`geometry.Vector`*).

The t must be a float between 0 (start of transition) and 1 (end of transition)

The `easing` must be an *`easings.Easing`* value.

```
a = 50
b = 100
t = 0.5
easing = Easing.none
result = ease_between(a, b, t, easing)
print('At time t={}, the value transitioning from a={} to b={} with easing {}␣
↪will be {}'.format(t, a, b, str(easing), result))
#  At time t=0.5, the value transitioning from a=50.0 to b=100.0 with easing␣
↪Easing.none will be 75.0
```

2.4 `engine` — Engine and Scenes: The core of your game

2.4.1 `Engine` reference

Constructor:

class engine.**Engine**(*virtual_resolution*, *virtual_resolution_mode=VirtualResolutionMode.adaptive_stretch*, *show_window=True*)

Engine instance is the first object you need to create to run the game.

Parameters:

- virtual_resolution - required. A *`geometry.Vector`* with width/height of the virtual resolution (see *virtual_resolution* for more information).
- virtual_resolution_mode - a *`VirtualResolutionMode`* value.
- show_window - if you pass False, the engine will start with a hidden window. Useful if you want to run kaa related stuff in a non-windowed environment, for example, when you want to run unit tests from a terminal window. Or when you want to start the game with a hidden window and show it manually later.

Game's 'entry point' is the *`Engine.run()`* method which takes in a *`Scene`* instance as a required parameter. Calling `run` will make the kaa engine run the scene, i.e. call its *`Scene.update()`* method in a loop.

A typical "Hello World" kaa game (showing just an empty window) would look like the following:

```
from kaa.engine import Engine, Scene
from kaa.geometry import Vector

class MyScene(Scene):
```

(continues on next page)

(continued from previous page)

```
    def update(self, dt):
        pass

with Engine(virtual_resolution=Vector(800, 600)) as engine:
    scene = MyScene()
    engine.run(scene)
```

To run the game in a fullscreen window, using 800x600 virtual resolution:

```
with Engine(virtual_resolution=Vector(800, 600)) as engine:
    scene = MyScene()
    engine.window.fullscreen = True
    engine.run(scene)
```

To run the game in a 1200x1000 window, using 800x600 resolution, without stretching the drawable area to fit the whole window size, giving window a title, and setting the clear color to green:

```
from kaa.engine import Engine, Scene, VirtualResolutionMode
from kaa.colors import Color
from kaa.geometry import Vector

with Engine(virtual_resolution=Vector(800, 600),
            virtual_resolution_mode=VirtualResolutionMode.no_stretch) as engine:

    scene = MyScene()
    engine.window.size = Vector(1200, 1000)
    engine.window.fullscreen = False
    engine.window.title = "Welcome to the wonderful world of kaa engine"
    scene.clear_color = Color(0, 1.0, 0, 1) # RGBA format
    engine.run(scene)
```

Be sure to check out the *virtual_resolution* documentation for more information on what virtual resolution concept is and how it is different than window size.

Instance properties:

Engine.**current_scene**
: Read only. Returns an active *Scene*. More complex games will have multiple scenes but the engine can run only one scene at a time. Only the active scene will have its `update()` method called by the engine.

  Use *Engine.change_scene()* method to change an active scene.

Engine.**virtual_resolution**
: Gets or sets the virtual resolution size. Expects *geometry.Vector* as a value, representing resolution's width and height.

  When writing a game you would like to have a consistent way of referencing coordinates, independent from the display resolution the game is running on. So for example when you draw some image on position (100, 200) you would like it to always be the same (100, 200) position on the 1366x768 laptop screen, 1920x1060 full HD monitor or any other of dozens display resolutions out there.

  That's where virtual resolution concept comes in. You declare a virtual resolution for your game just once, when initializing the engine, and the engine will always use exactly this resolution when you draw stuff in your game. If you run the game in a window larger than the declared virtual resolution, the engine will stretch the game's actual draw area. If you run it in a window smaller than declared virtual resolution, the engine will shrink it.

  There are different policies available for stretching and shrinking the area. You can control it by setting the *virtual_resolution_mode* property.

Although it is possible to change the virtual resolution (even as the game is running), we don't recommend it unless you have a good reason to do that.

Engine.**virtual_resolution_mode**
Gets or sets virtual resolution mode. See *VirtualResolutionMode* documentation for a list of possible values.

It is possible to change the virtual resolution mode, even as the game is running.

```
from kaa.engine import get_engine, VirtualResolutionMode

engine = get_engine()
engine.virtual_resolution_mode = VirtualResolutionMode.aggresive_stretch
```

Engine.**window**
A get accessor to the *Window* object which exposes game window properties such as window size, title, or fullscreen flag and allows to change them.

Note: It is perfectly safe to change the window size or fullscreen mode, even in the game runtime.

Check out the *Window* documentation for a list of all available properties and methods.

```
from kaa.engine import get_engine
from kaa.geometry import Vector

engine = get_engine()
engine.window.title = "Hello world"
engine.window.fullscreen = False
engine.window.size = Vector(1920, 1080)
```

Engine.**audio**
A get accessor to the *AudioManager* object which exposes global audio properties such as the master volume for sound effects or music. See the *AudioManager* documentation for a list of all available properties.

```
from kaa.engine import get_engine

engine = get_engine()
engine.audio.master_sound_volume = 0.5 # 50% of the max volume (sfx)
engine.audio.master_music_volume = 0.75 # 75% of the max volume (music)
engine.audio.mixing_channels = 100 # set number of max sounds we'll be able to␣
↪play simultaneously
```

Instance methods:

Engine.**change_scene**(*new_scene*)
Use this method to change the active scene. Only one scene can be active at a time.

Active scene is being rendered by the renedrer and has its `update()` method called.

A non-avtive scene remains 'frozen': it does not lose state (no objects are ever removed by deactivating a Scene) but its `update()` method is not being called and nothing is being rendered.

Example of having two scenes and toggling between them:

```
from kaa.input import Keycode
from kaa.engine import Engine, Scene
from kaa.colors import Color
from kaa.geometry import Vector
```

(continues on next page)

(continued from previous page)

```
from kaa.fonts import TextNode, Font
import os

SCENES = {}

class TitleScreenScene(Scene):

    def __init__(self, font):
        super().__init__()
        self.root.add_child(TextNode(font=font, font_size=30, position=Vector(500,
↪ 500),
                                     text="This is the title screen, press enter␣
↪to start the game.",
                                     color=Color(1, 1, 0, 1)))

    def update(self, dt):
        for event in self.input.events():
            if event.keyboard_key:
                if event.keyboard_key.is_key_down and event.keyboard_key.key ==␣
↪Keycode.return_:
                    self.engine.change_scene(SCENES['gameplay_scene'])


class GameplayScene(Scene):

    def __init__(self, font):
        super().__init__()
        self.label = TextNode(font=font, font_size=30, position=Vector(1000, 500),
↪ color=Color(1, 0, 0, 1),
                              text="This is gameplay, press q to get back to the␣
↪title screen. I'm rotating BTW.")
        self.root.add_child(self.label)

    def update(self, dt):
        for event in self.input.events():
            if event.keyboard_key:
                if event.keyboard_key.is_key_down and event.keyboard_key.key ==␣
↪Keycode.q:
                    self.engine.change_scene(SCENES['title_screen_scene'])
        self.label.rotation_degrees += dt * 20


with Engine(virtual_resolution=Vector(1920, 1080)) as engine:
    font = Font(os.path.join('assets', 'fonts', 'DejaVuSans.ttf'))  # MUST create␣
↪all kaa objects inside engine context!
    SCENES['title_screen_scene'] = TitleScreenScene(font)
    SCENES['gameplay_scene'] = GameplayScene(font)
    engine.window.fullscreen = True
    engine.run(SCENES['title_screen_scene'])
```

Engine.**get_displays**()

Returns a list of all available *Display* objects (monitors) present in the system. See the *Display* documentation for a full list of display properties avaiable.

```
from kaa.engine import get_engine
```

(continues on next page)

(continued from previous page)

```
engine = get_engine()
for display in engine.get_displays():
    print(display)
```

Engine.**quit**()
Destroys the engine and closes the window. Call this method when the player wants to leave the game or to handle the quit event received from the system on closing the window (e.g. by ALT+F4 or pressing "X")

```
from kaa.engine import Scene
from kaa.input import Keycode

class MyScene(Scene):

    def update(self, dt):

        for event in self.input.events():
            if event.system and event.system.quit:
                # handle the system event of pressing "X" or ALT+F4 to close the␣
↪window:
                self.engine.quit()

            if event.keyboard_key and event.keyboard_key.key == Keycode.q:
                # quit the game on pressing the Q key
                self.engine.quit()
```

Engine.**run**(*scene*)
Starts running a scene instance, by calling its `update` method in a loop. You'll need to call this method just once, to run the first scene of your game. To change between scenes use the *Engine.change_scene()* method.

Engine.**stop**()
Stops the engine. You won't need to call it if you use context manager, i.e. initialize the Engine using the `with` statement.

Engine.**get_fps**()
Returns current frames per second rate. It is an average from the last 10 frames.

2.4.2 Scene reference

Constructor:

class engine.**Scene**
The Scene instance is a place where all your in-game objects will live. You should write your own scene class by inheriting from this type. Scene main features are:

- Each Scene must define a *Scene.update()* method which will be called by the engine on every frame.
- Use the *nodes.Node.add_child()* method on Scene's *root node* to add objects (Nodes) to the Scene. *Read more about Nodes*.
- Use the *input* property to access *input.InputManager* which:
  - exposes a lot of methods to actively check for input from mouse, keyboard, controllers etc.
  - includes an events list which occurred during the current frame (mouse, keyboard, controllers, music, etc.)
- Use the *camera* property to control the camera

- Use the *views* property to access views. Read more how *View* objects work.

The Scene constructor does not take any parameters. As stated above, you should never instantiate a *Scene* directly but write your own scene class that inherit from it. Use the Scene's constructor to add initial objects to the scene

```
from kaa.engine import Scene

def MyScene(Scene):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # initialize the scene here, e.g. add some initial objects to the scene...

    def update(self, dt):
        pass

with Engine(virtual_resolution=Vector(800, 600)) as engine:
    scene = MyScene()
    engine.run(scene)
```

Instance properties:

Scene.**camera**
A get accessor to the *Camera* object of the default view (*read more about views here*). Camera object includes properties and methods for manipulating the camera (moving, rotating, etc.). See the *Camera* documentation for a full list of available properties and methods.

```
from kaa.engine import Scene
from kaa.geometry import Vector

def MyScene(Scene):

    def __init__(self):
        self.camera.position = Vector(-200, 400)
        self.camera.rotation_degrees = 45
        self.camera.scale = Vector(2.0, 2.0)
```

Scene.**engine**
Returns *Engine* instance.

Scene.**views**
Allows for accessing views by index. Each Scene has 32 views. Check out *View* reference for more information on how views work.

```
def MyScene(Scene):

    def how_to_access_views(self)
        print(len(self.views))  # 32 views
        the_default_view = self.views[0] # view with 0 index is the default view.
        some_view = self.views[17]
```

Scene.**input**
A get accessor to the *input.InputManager* object which offers methods and properties to actively check for input from mouse, keyboard, controllers etc. It also allows to consume events published by those devices, by the system or by the kaa engine itself. Check out the *input.InputManager* documentation for a full list of available features.

```
from kaa.engine import Scene
from kaa.geometry import Vector
from kaa.input import Keycode, MouseButton

def MyScene(Scene):

    def update(self, dt):

        # actively check if a "W" key is pressed
        if self.input.is_pressed(Keycode.w):
            # .... do something
        # consume all events that occurred during the frame:
        for event in self.input.events():
            # .... do something
```

Scene.**root**

All objects which you will add to the scene (or remove from the scene) are called Nodes. Nodes can form a tree-like structure, that is: a Node can have many child Nodes, and exacly one parent Node. Each Scene has a "root" node, accessible by this property.

Refer to the *nodes* documentation for more information on how the nodes work.

```
from kaa.engine import Scene
from kaa.nodes import Node
from kaa.sprites import Sprite

def MyScene(Scene):

    def __init__(self):
        super().__init__()
        self.arrow_sprite = Sprite(os.path.join('assets', 'gfx', 'arrow.png'))
        self.arrow_node = Node(sprite=self.arrow_sprite, position=Vector(200,␣
↪200))
        self.root.add_child(self.arrow_node)
```

Scene.**clear_color**

Gets or sets the clear color (*colors.Color*) for the default view. Check out *View* documentation for more information on views.

An example of 800x600 viewport, colored in green, running in the 1200x1000 window using `no_stretch` mode:

```
from kaa.engine import Engine, Scene, VirtualResolutionMode
from kaa.colors import Color
from kaa.geometry import Vector

class MyScene(Scene):

    def update(self, dt):
        pass


with Engine(virtual_resolution=Vector(800, 600),
            virtual_resolution_mode=VirtualResolutionMode.no_stretch) as␣
↪engine:

    scene = MyScene()
    scene.clear_color = Color(0, 1, 0, 1) # RGBA format
```

(continues on next page)

(continued from previous page)

```
    engine.window.size = Vector(1200, 1000)
    engine.run(scene)
```

`Scene.`**`spatial_index`**
A get accessor to the *SpatialIndexManager*, which offers methods to query for nodes at specific position or inside a specific *geometry.BoundingBox*

`Scene.`**`time_scale`**
Gets or sets a time scale, as float. Kaa engine will apply this scale everywhere (in physics, timers, transitions and so on). Basically it allows you to speed up or slow down the time scale of your whole game.

Instance methods:

`Scene.`**`update`**`(`*dt*`)`
An update method is called every frame. The dt parameter is a time elapsed since previous update call, in seconds. Most of your game logic will live inside the update method.

Note: If you change the *Scene.time_scale* value the dt value received by the update() will be adjusted accordingly.

```
from kaa.engine import Scene
from kaa.nodes import Node
from kaa.sprites import Sprite

def MyScene(Scene):

    def __init__(self):
        super().__init__()
        self.arrow_sprite = Sprite(os.path.join('assets', 'gfx', 'arrow.png'))
        self.arrow_node = Node(sprite=self.arrow_sprite, position=Vector(200,␣
↪200))
        self.root.add_child(self.arrow_node)

    def update(self, dt)
        self.arrow_node.rotation_degrees += 20 * dt # rotate the arrow 20 degrees␣
↪per second, clockwise
```

`Scene.`**`on_enter`**`()`
This method is called when the scene is activated (either by *Engine.run()*, or by *Engine.change_scene()*) giving you opportunity to write some logic each time that happens.

`Scene.`**`on_exit`**`()`
Same as *Scene.on_enter()* but is called just before the scene gets deactivated via the *Engine.change_scene()*.

2.4.3 `SpatialIndexManager` reference

`class` `engine.`**`SpatialIndexManager`**
Input manager object can be accessed via *Scene.spatial_index* property. It has two main features:

- Find Nodes on the Scene, at given position (x,y)
- Find Nodes on the Scene inside a specific *geometry.BoundingBox*.

Note that only nodes whose `indexable` property is set to True will be queried.

Instance methods:

`SpatialIndexManager.`**`query_bounding_box`**(*bounding_box*, *include_shapeless=True*)

Returns a list of Nodes inside specified bounding box. It also includes those which only intersect the bounding box. The `bounding_box` must be an instance of *geometry.BoundingBox*. Returned nodes are unordered.

The `include_shapeless` param determines whether the query will also include nodes which do not have a *shape*.

Note: Only the nodes with indexable property set to True will be queried. The indexable property is True by default.

```
from kaa.geometry import BoundingBox

nodes = scene.spatial_index.query_bounding_box(BoundingBox(100, 150, 500, 600))
print("found {} nodes inside or intersecting that bounding box!".
↪format(len(nodes)))
```

`SpatialIndexManager.`**`query_point`**(*point*)

Returns a list of Nodes that contain the specified point. The `point` must be a *geometry.Vector*. Returned nodes are unordered.

Note: Only the nodes with indexable property set to True will be queried. The indexable property is True by default.

```
from kaa.geometry import Vector

nodes = scene.spatial_index.query_point(Vector(100, 150))
print("found {} nodes which contain that point!".format(len(nodes)))
```

2.4.4 `View` reference

`class` `engine.`**`View`**

Views allow you to fine-tune how the scene is being redenred on the screen. Each scene has 32 views (indexed -16 to 15) which you can configure independently. You can configure a view to be displayed at given position on the screen, give it specific width/height and then use the view's camera to show the scene normally inside the view's box. Important caveat: **the view will render only Nodes which were explicitly assigned to it.**. It means you need to use the *views property on a Node* to explicitly assign a Node to a specific view.

Each view's index determines its z_index property. It is used to manage the 'layering' of all 32 views. In other words all nodes assigned to view with a higher index will be rendered in front of any node assigned to a view with a lower index, regardless of node's z_index values. Node's z_index values are used to manage 'layering' of nodes **within a view**.

An example below configures a 400x400 view at position (100, 200) and inside that box it displays a fragment of the scene using view's camera:

```
def MyScene(Scene):

    def see_what_views_have_to_offer(self):
        some_view = self.views[1]  # Note: there are 32 views available
        some_view.origin = Vector(100, 200)  # The view will be positioned at␣
↪(100, 200), in relation to display
```

(continues on next page)

(continued from previous page)

```
        some_view.dimensions = Vector(400, 400) # view 'box' size will be 400x400
        # The view has its own camera which you can manage normally to show the␣
↪scene inside the box:
        some_view.camera.position = Vector(300, 500)
        some_view.camera.scale = Vector(3, 3)
        some_view.clear_color = Color(1, 0, 0, 1)  # we can set the clear color␣
↪for the view as well

        # we may add nodes to views independently.
        self.root.add_child(Node())  # will be rendered in the default view only␣
↪(0)
        self.root.add_child(Node(views={0, 1, 2, 14}))  # the node will be␣
↪rendered in views 0, 1, 2, 14
```

Few typical use cases for views:

- Build a UI layer (panels, buttons, menus etc.) - add all those nodes to a separate view
- Split screen feature - render scene in multiple views each covering part of the screen, cameras focused on different players
- Makes parallax scrolling effect easier to implement - render each layer using separate view.

Instance properties:

`View.`**`origin`**
Gets or sets the origin of a view, as *`geometry.Vector`*. The origin points to the top-left position of the view on the screen.

`View.`**`dimensions`**
Gets or sets the dimensions of a view, as *`geometry.Vector`*, x being width and y being height.

`View.`**`clear_color`**
Gets or sets a clear color for a view as *`colors.Color`*. Default color is black.

`View.`**`camera`**
Returns a *`Camera`* associated with this view.

`View.`**`z_index`**
Read only. Gets the z_index of a view, that is basically view index (can be a value between -15 and 16)

2.4.5 `Window` reference

`class` `engine.`**`Window`**

Window object exposes properties and methods for the game window. Changing the `fullscreen` flag will make the game run in a fullscreen or windowed mode. If you run the game in the windowed mode, you can resize or reposition the window using properties such as `position`, `size` or methods such as *`Window.center()`*.

Instance properties:

`Window.`**`fullscreen`**

Gets or sets the fullscreen mode. Expects bool value. Setting fullscreen to `True` will remove the window's borders and title bar and stretch it to fit the entire screen.

It is possible to toggle between fullscreen and windowed mode as the game is running.

```
from kaa.engine import get_engine

engine = get_engine()
engine.window.fullscreen = True
```

Window.**size**

Gets or sets the size of the window, using *geometry.Vector*.

Note that if you set the `fullscreen` to `True` the window will not only resize automatically to fit the entire screen but will also drop its borders and the top bar. Resizing the window programatically makes most sense if the game already runs in the windowed mode (with `window.fullscreen=False`).

```
from kaa.engine import get_engine
from kaa.geometry import Vector

engine = get_engine()
engine.window.size = Vector(500, 300)  # sets the window size to 500x300
```

Window.**position**

Gets or sets the position of the window on the screen, using *geometry.Vector*. Passing Vector(0,0) will align the window with the top left corner of the screen.

Just like with the `size` attribute, changing window position makes sense only if using windowed mode (`window.fullscreen=False`).

```
from kaa.engine import get_engine
from kaa.geometry import Vector

engine = get_engine()
engine.window.position = Vector(0, 0)
```

Window.**title**

Gets or sets the title of the window.

```
from kaa.engine import get_engine

engine = get_engine()
engine.window.title = "Git Gud or Get Rekt!"
```

Instance methods:

Window.**center**()
: Positions the window in the center of the screen. Makes most sense if using windowed mode (`window.fullscreen=False`)

Window.**maximize**()
: Maximizes the window. Makes most sense if using windowed mode (`window.fullscreen=False`)

Window.**minimize**()
: Minimizes the window. Makes most sense if using windowed mode (`window.fullscreen=False`)

Window.**show**()
: Shows the window.

Window.**hide**()
: Hides the window.

Window.**restore**()
Restores the window from the maximized/minimized state to the default state. Makes most sense if using windowed mode (`window.fullscreen=False`)

2.4.6 `AudioManager` reference

class engine.**AudioManager**

Audio Manager gives access to global audio settings, such as master sound volume. Audio Manager can be accessed via the *Engine.audio* property on the Engine instance.

Instance properties:

AudioManager.**master_volume**

Gets or sets the master volume level for sounds and music, using value between 0 (0% volume) and 1 (100% volume).

Master volume affects sound effects and music tracks volume played with *audio.Sound.play()* and *audio.Music.play()* respectively.

```
from kaa.engine import get_engine

# somwhere inside Scene....
self.engine.master_volume = 1.0  # sets master volume to 100%
my_sound.play(volume=0.7)  # plays a sound with 70% volume
self.engine.master_volume = 0.1  # sets master volume to 10%
my_sound.play(volume=0.5)  # pays a sound with 5% volume (50% sound volume * 10%␣
↪master volume = 5% final volume)
```

AudioManager.**master_sound_volume**

Gets or sets the default volume level for sound effects. Using value between 0 (0% volume) and 1 (100% volume). Master sound volume level affects sound effects volume played with *audio.Sound.play()*

```
from kaa.engine import get_engine

# somwhere inside Scene class ....
self.engine.master_sound_volume = 1.0  # sets master sfx volume to 100%
my_sound.play(volume=0.7)  # plays a sound with 70% volume
self.engine.master_volume = 0.1  # sets master volume to 10%
my_sound.play(volume=0.5)  # pays a sound with 5% volume (50% sound volume * 10%␣
↪master sfx volume = 5% final volume)
```

AudioManager.**master_music_volume**

Gets or sets the default master volume level for music. Using value between 0 (0% volume) and 1 (100% volume). Master music volume level affects music tracks volume played with *audio.Music.play()*.

```
from kaa.engine import get_engine

# somwhere inside Scene....
self.engine.master_music_volume = 1.0  # sets master music volume to 100%
my_music.play(volume=0.7)  # plays a music track with 70% volume
self.engine.master_music_volume = 0.1  # sets master music volume to 10%
my_music.play(volume=0.5)  # pays music track with 5% volume (50% sound volume * 10%␣
↪master music volume = 5% final volume)
```

AudioManager.**mixing_channels**

Gets or sets the maximum number of sound effects that can be played simultaneously with *audio.Sound.play()*. Note that you can never play more than one music track simultaneously.

2.4.7 `Display` reference

class engine.**Display**

Stores display device properties. A list of Display objects can be obtained by calling *Engine.get_displays()*.

Instance Properties:

Display.**index**

Read only. Returns display index (integer).

Display.**name**

Read only. Returns display name.

Display.**position**

Read only. Returns display position as *geometry.Vector*.

Display.**size**

Read only. Returns display resolution as *geometry.Vector*.

2.4.8 `Camera` reference

class engine.**Camera**

A camera projects the image of the 2D scene onto the screen. You can move, rotate or scale the camera by setting its properties.

To get a Camera instance, either

1) use the *Scene.camera* property which returns a default camera, or
2) Access a specific view on a Scene via *Scene.views* and then access the camera property via *View.camera*

Note: There isn't a "global" camera - each *View* has its own camera allowing to display fragments of scene in different viewports.

Instance properties:

Camera.**position**
: Gets or sets the camera position, using *geometry.Vector*.

```
from kaa.geometry import Vector

# somewhere inside Scene:
self.camera.position = Vector(123.45, 678.9)
```

Camera.**rotation**
: Gets or sets the camera rotation, in radians

```
from kaa.geometry import Vector
import math

# somewhere inside Scene:
self.camera.rotation = math.pi / 4
```

Camera.**rotation_degrees**
Gets or sets the camera rotation, in degrees

```
from kaa.geometry import Vector
import math

# somewhere inside Scene:
self.camera.rotation_degrees = 180 # show the scene upside down
```

Camera.**scale**
Gets or sets the scale for the camera (using *geometry.Vector*). In other words, manipulating this property allows for a zoom-in / zoom-out effects. Each axis (x and y) can be manipulated independently, so if you zoom in on X axis and zoom out on Y the image projected by the camera will appear stretched.

```
from kaa.geometry import Vector
import math

# somewhere inside Scene:
self.camera.scale= Vector(1.5, 1.5) # 50% zoom-in
```

Camera.**visible_area_bounding_box**
Returns camera's visible area as *geometry.BoundingBox*.

Instance methods:

Camera.**unproject_position**(*position*)

Takes a position (*geometry.Vector*), applies all camera transformations (position, scale, rotation) to that position and returns the result. Useful when you want to convert position in the screen frame reference (as returned by MouseEvent-Button.position) or when you have applied some transformations to the camera and want to know the actual position of given point (e.g. mouse position)

Full example:

```
import os
from kaa.engine import Engine, Scene
from kaa.geometry import Vector
from kaa.input import MouseButton
from kaa.fonts import TextNode, Font


class MyScene(Scene):

    def __init__(self, font):
        self.root.add_child(TextNode(font=font, font_size=30,␣
↪position=Vector(400, 300), z_index=10,
            text="This is a static text, it never rotates itself. Click to␣
↪rotate the camera 45 degrees"))

    def update(self, dt):

        for event in self.input.events():
```

(continues on next page)

(continued from previous page)

```
            if event.mouse_button and event.mouse_button.is_button_down and␣
↪event.mouse_button.button == MouseButton.left:
                position = self.input.mouse.get_position()
                unproj_position = self.camera.unproject_position(position)
                print(f'Before the camera rotation: Mouse position {position}
↪ -> unproject -> {unproj_position}')
                # let's now rotate the camera 45 degrees and check the result
                self.camera.rotation_degrees += 45
                unproj_position = self.camera.unproject_position(position)
                print(f'After camera rotation: Mouse position {position} ->␣
↪unproject -> {unproj_position}')


with Engine(virtual_resolution=Vector(800,600)) as engine:
    font = Font(os.path.join('assets', 'fonts', 'DejaVuSans.ttf'))
    engine.run(MyScene(font))
```

2.4.9 `VirtualResolutionMode` reference

class engine.**VirtualResolutionMode**

VirtualResolutionMode is an enum type which you can pass when creating the *engine.Engine* instance.

It tells the engine how it should stretch the virtual resolution (set via the `virtual_resolution` property).

- `VirtualResolutionMode.adaptive_stretch` - the default mode. The drawable area will adapt to window size, maintaining aspect ratio and leaving black padded areas outside
- `VirtualResolutionMode.aggresive_stretch` - the drawable area will always fill the entire window - aspect ratio may not be maintained while stretching.
- `VirtualResolutionMode.no_stretch` - no stretching applied, leaving black padded areas if window is larger than virtual resolution size

2.4.10 `get_engine()` reference

engine.**get_engine**()

This function provides a convenient way of getting an engine instance from anywhere in your code.

```
from kaa.engine import get_engine

engine = get_engine()
```

2.5 `fonts` — Drawing text on screen

2.5.1 `Font` reference

Constructor:

class fonts.**Font**(*font_filepath*)

Font object is used to load a font from a file. Font objects are immutable.

The Font constructor accepts just one parameter: `font_filepath` which should be a string with a path to a font file.

Kaa engine currently supports the following font file formats:

- ttf

```
import os
from kaa.fonts import Font

# .... somewhere inside a Scene ...
font = Font(os.path.join('assets','fonts','DejaVuSans.ttf'))
```

2.5.2 `TextNode` reference

Constructor:

class fonts.**TextNode**(*font*, *text=""*, *font_size=28.0*, *line_width=float("Inf")*, *interline_spacing=1.0*, *first_line_indent=0*, *position=Vector(0, 0)*, *rotation=0*, *scale=Vector(1, 1)*, *z_index=0*, *color=Color(0, 0, 0, 0)*, *sprite=None*, *shape=None*, *origin_alignment=Alignment.center*, *lifetime=None*, *transition=None*, *visible=True*)

TextNode extends the *nodes.Node* class to give you ability to comfortably work with text.

In addition to all *nodes.Node* params the TextNode constructor accepts the following ones:

- `font` - a *Font* instance
- `font_size` - a number
- `line_width` - a number
- `interline_spacing` - a number
- `first_line_indent` - a number

```
class MyScene(Scene):

    def __init__(self):
        font = Font(os.path.join('assets','fonts','DejaVuSans.ttf'))

        # a simple label
        simple_text = TextNode(font=font, text="Hello world", position=Vector(100,
↪ 100), font_size=30,
                               origin_alignmnet=Alignment.left, color=Color(1,1,0,
↪1), z_index=100)

        # a paragraph with a width of 300 and first line indent of 50. Words will␣
↪wrap automatically
        wrapped_text = TextNode(font=font,
                                text="Lorem ipsum dolor sit amet, consectetur␣
↪adipiscing elit. Ut dignissim, tellus "
                                     "sit amet ultrices facilisis, purus mi␣
↪malesuada ante, sit amet ultricies erat "
                                     "mauris a turpis. Integer a elit sed mi␣
↪mattis tincidunt. Pellentesque tristique "
                                     "semper cursus. Maecenas suscipit, ex quis␣
↪condimentum consectetur, quam sapien "
                                     "placerat ex, eu aliquam est est condimentum␣
↪mauris. ",
```

(continues on next page)

(continued from previous page)

```
                                position=Vector(500, 500), font_size=30,
                                origin_alignmnet=Alignment.center, color=Color(1,␣
↪0, 0, 1), line_width=300,
                                first_line_indent=50, z_index=101)

        self.root.add_child(simple_text)
        self.root.add_child(wrapped_text)
```

Instance properties

`TextNode.`**`text`**
Gets or sets a text to be rendered. A string.

Note: Updating text is relatively heavy operation in terms of performance so you should avoid doing it on each frame on a large number of nodes.

`TextNode.`**`font_size`**
Gets or sets the font size to be used when rendering the text. A number. Default is 28.

`TextNode.`**`line_width`**
Gets or sets the paragraph width. A number. Words will wrap automatically to fit the desired width. Default is infinite width.

`TextNode.`**`interline_spacing`**
Gets or sets the spacing between the lines of text in case of multiline texts.

`TextNode.`**`first_line_indent`**
Gets or sets the first line indentation in case of multiline texts.

2.6 `geometry` — wrapper classes for vectors, segments, polygons etc.

2.6.1 `Vector` reference

Constructor:

`class` `geometry.`**`Vector`**(*x*, *y*)
Vector instance represents an Euclidean vector. It stores a pair od 2D corrdinates (x, y).

Vectors are immutable.

Vectors are used for the following purposes:

- storing an actual vector pointing from (0, 0) to (x, y), for example `nodes.BodyNode.velocity`
- storing a 2D point, for example *`nodes.Node.position`*
- storing a width/height of a rectangular shape, such as a screen resolution. For example *`engine.Engine.virtual_resolution`*

Vector constructor accepts two float numbers: x and y.

Available operators:

- Adding two vectors: `Vector(1,1) + Vector(2,2)`
- Substracting two vectors: `Vector(1,1) - Vector(2,2)`

- Multiplying vector by a scalar: `Vector(1,1) * 123`
- Dividing vector by a scalar: `Vector(1,1) / 123`

Class methods:

classmethod `Vector.`**from_angle**(*angle*)
Creates a new unit Vector (i.e. length 1 vector) from angle, in radians.

```
import math
from kaa.geometry import Vector

v = Vector.from_angle(math.pi / 4)
print(v)  # V[0.7071067811865476, 0.7071067811865475]
print(v.length())  # 1.0
```

classmethod `Vector.`**from_angle_degrees**(*degrees*)
Creates a new unit Vector (i.e. length 1 vector) from angle, in degrees.

```
import math
from kaa.geometry import Vector

v = Vector.from_angle_degrees(90) # 90 degrees is pointing up, 180, pointing left,
↪ 270 pointing down etc.
print(v)  # V[0.0, 1.0]
print(v.length())  # 1.0
```

Instance Properties (read only):

`Vector.`**x**
Gets the x value of a vector

`Vector.`**y**
Gets the y value of a vector

Instance Methods:

`Vector.`**is_zero**()
Returns `True` if vector is a zero vector

```
from kaa.geometry import Vector

Vector(0, 0).is_zero()  # True
Vector(0.1, 0).is_zero()  # False
```

`Vector.`**rotate_angle**(*angle*)
Returns a new vector, rotated by given angle, in radians.

```
from kaa.geometry import Vector
import math

print(Vector(10, 0))  # V[10, 0]
print(Vector(10, 0).rotate_angle(math.pi))  # V[-10, 0]
```

`Vector.`**rotate_angle_degrees**(*degrees*)
Returns a new vector, rotated by given angle, in degrees.

```
from kaa.geometry import Vector
import math

```

(continues on next page)

(continued from previous page)

```
print(Vector(10, 0))  # V[10, 0]
print(Vector(10, 0).rotate_angle_degrees(180))  # V[-10, 0]
```

`Vector.`**`to_angle`**`()`
Returns vector's angle, in radians.

`Vector.`**`to_angle_degrees`**`()`
Returns vector's angle, in degrees.

`Vector.`**`dot`**`(`*other_vector*`)`
Returns dot product of two vectors. other_vector parameter must be *geometry.Vector*

`Vector.`**`distance`**`(`*other_vector*`)`
Returns a distance from (x,y) to (other_vector.x, other_vector.y), in other words: distance between two points. other_vector parameter must be *geometry.Vector*

`Vector.`**`angle_between`**`(`*other_vector*`)`
Returns angle between this vector and `other_vector`, in radians. The other_vector parameter must be *geometry.Vector*

`Vector.`**`angle_between_degrees`**`(`*other_vector*`)`
Returns angle between this vector and `other_vector`, in degrees. The other_vector parameter must be *geometry.Vector*

`Vector.`**`normalize`**`()`
Returns a new vector, normalized (i.e. unit vector)

`Vector.`**`length`**`()`
Returns vector's length.

2.6.2 `Segment` reference

Constructor:

`class` `geometry.`**`Segment`**`(`*vector_a*, *vector_b*`)`
Segment instance represents a segment between two points, a and b.

Segments are immutable.

vector_a and vector_b params are *geometry.Vector* instances indicating both ends of a Segment

Instance properties:

`Segment.`**`point_a`**
Read only. Returns point A of the segment

`Segment.`**`point_b`**
Read only. Returns point B of the segment

`Segment.`**`bounding_box`**
Read only. Returns segment's bounding box as *geometry.BoundingBox*.

Instance methods:

`Segment.`**`transform`**`(`*transformation*`)`
Applies given transformation to this Segment and returns a new Segment.

The `transformation` parameter must be a *Transformation* instance.

2.6.3 `Circle` reference

Constructor:

class `geometry.`**Circle**(*radius*, *center=Vector(0, 0)*)

Circle instance represents a circualar shape, with a center and a radius. Circles are used e.g. for creating hitboxes.

Circles are immutable.

The `center` parameter must be *`geometry.Vector`*, radius is a number.

Instance properties:

`Circle.`**radius**

Read only. Returns circle radius.

`Circle.`**center**

Read only. Returns circle center.

`Circle.`**bounding_box**

Read only. Returns circle bounding box as *`geometry.BoundingBox`*.

Instance methods:

`Circle.`**transform**(*transformation*)

Applies given transformation to this Circle and returns a new Circle.

The `transformation` parameter must be a *`Transformation`* instance.

2.6.4 `Polygon` reference

Constructor:

class `geometry.`**Polygon**(*points*)

Polygon instance represents a custom shape. Polygons are used e.g. for creating hitboxes.

Polygons are immutable.

The `points` parameter must be a list of *`geometry.Vector`* instances.

If you don't close the polygon (the last point in the list is not identical with the first one) kaa will do that for you.

The polygon must be convex. Kaa engine will throw an exception if you try to create a non-convex polygon. You may use *`classify_polygon()`* function to check if a list of points will form a convex polygon or not.

```
from kaa.geometry import Polygon

polygon = Polygon([Vector(-10, -10), Vector(10, 30), Vector(0, 40)])  # a␣
↪triangular-shaped polygon
```

Class methods:

classmethod `Polygon.`**from_box**(*vector*)

Creates a rectangular-shaped Polygon whose central point is at (0, 0) and width and height are passed as vector.x and vector.y respectively. A useful shorthand function for creating a rectangular shape for a *`physics.HitboxNode`*.

```
from kaa.geometry import Polygon, Vector

poly = Polygon.from_box(Vector(10, 8)) # creates a rectangular polygon [ V(-5, -
↪4), V(5, -4), V(5, 4), V(-5, 4) ]
```

Instance properties:

`Polygon.`**`points`**
Read only. Returns a list of points constituting the Polygon.

`Polygon.`**`bounding_box`**
Read only. Returns polygon's bounding box as *`geometry.BoundingBox`*.

Instance methods:

`Polygon.`**`transform`**(*transformation*)
Applies given transformation to this Polygon and returns a new Polygon.

The `transformation` parameter must be a *`Transformation`* instance.

2.6.5 `Transformation` reference

`class` `geometry.`**`Transformation`**
Transformation is a 'geometrical recipe', which can be applied to a *`Segment`*, *`Circle`* or *`Polygon`* (using the `transform()` method) to change their position, rotation and scale.

Transformations cannot be applied to *Nodes*, although if a Node *has a shape*, you can apply Transformations to that shape.

Transformation objects are immutable.

Transformation constructor does not accept any parameters and creates a 'void' transformation which, when applied, does not have any effect.

To create an actual Transformation use one of the class methods: *`rotate()`*, *`rotate_degrees()`*, *`scale()`* or *`translate()`*

```
from kaa.geometry import Transformation
import math

t1 = Transformation.rotate(math.pi / 2)  # a 90 degrees transformation, clockwise
t2 = Transformation.rotate_degrees(-45)  # a 45 degrees transformation, anti-
↪clockwise
t3 = Transformation.scale(Vector(2,2))  # scale change transformation (enlarge␣
↪twice)
t4 = Transformation.translate(Vector(10, 0))  # position change transformation␣
↪(10 units to the right)
```

You can chain transformations by applying the `|` operator, which results in a new, combined transformations:

```
combined_transformation = t1 | t2 | t3 | t4
```

Rotation and scaling is always relative to the origin of the Euclidean space, or in other words, relative to (0,0) point. Therefore, a sequence of transformations in a chain is important. Consider the following two transformations:

```
rotate_than_move = t2 | t4
move_than_rotate = t4 | t2
```

Contrary to intuition they won't give the same result. When applied to a square with an edge length of 1 and the middle in the (0,0) the first one will rotate the square 45 degrees around (0,0) and then move 10 units to the right, while the second one will move the square 10 units to the right and then rotate, but since the center of the square is now at (10,0) the rotation is going to "wheel" it 45 degrees around the (0,0), making the Polygon end up in a different position. It's illustrated in the example below:

```
from kaa.geometry import Vector, Polygon

square = Polygon.from_box(Vector(2,2))
print(square.points)  #[V[-1.0, -1.0], V[1.0, -1.0], V[1.0, 1.0], V[-1.0, 1.0]]

# just move it
square_2 = square.transform(t4)
print(square_2.points)  #[V[9.0, -1.0], V[11.0, -1.0], V[11.0, 1.0], V[9.0, 1.0]]

# rotate then move
square_3 = square.transform(t2 | t4)
print(square_3.points)  # [V[8.585, 0.0], V[10.0, -1.414], V[11.414, 0.0], V[10.0,
↪ 1.414]]

# move then rotate
square_4 = square.transform(t4 | t2)
print(square_4.points)  # [V[5.656, -7.071], V[7.071, -8.485], V[8.485, -7.071],␣
↪V[7.0710, -5.656]]
```

Using the @ operator you can chain transformation in the matrix-style order:

```
rotate_then_move = t2 | t4
move_then_rotate = t2 @ t4
```

Finally, you can use the *inverse()* method on the Transformation instance to get the inversed transformation:

```
combined_transformation = t1 | t2 | t3 | t4
inversed_combined_transformation = combined_transformation.inverse()
```

Class methods:

classmethod Transformation.**rotate**(*rotation*)
Creates a new rotation Transformation. The rotation value must be a number (rotation in radians).

classmethod Transformation.**rotate_degrees**(*rotation_degrees*)
Creates a new rotation Transformation. The rotation value must be a number (rotation in degrees).

classmethod Transformation.**scale**(*scaling_vector*)
Creates a new scaling Transformation. The scaling_vector must be a *Vector* whose x and y represent scaling in x and y axis respectively.

classmethod Transformation.**translate**(*translation_vector*)
Creates a new translation (position change) Transformation. The translation_vector must be a *Vector*.

Instance methods:

Transformation.**inverse**()
Returns a new Transformation, being an inversed version of this Transformation.

Transformation.**decompose**()
Returns a *DecomposedTransformation* object which allows reading transformation's translation, rotation and scale.

```
combined_transformation = t1 | t2 | t3 | t4
result = combined_transformation.decompose()
print(result.translation, result.rotation, result.rotation_degrees, result.scale)
```

2.6.6 `DecomposedTransformation` reference

class geometry.**DecomposedTransformation**

Object returned by *Transformation.decompose()*. It surfaces transformation properties.

Instance properties:

DecomposedTransformation.**translation**
Returns translation as *geometry.Vector*

DecomposedTransformation.**rotation**
Returns rotation as float, in radians

DecomposedTransformation.**rotation_degrees**
Returns rotation as float, in degrees

DecomposedTransformation.**scale**
Returns scale, as *geometry.Vector*

2.6.7 `BoundingBox` reference

class geometry.**BoundingBox**(*min_x*, *min_y*, *max_x*, *max_y*)
Represents a rectangular bounding box. Bounding box is always aligned with x and y axis. Bounding boxes are being used when querying for nodes on scene. Constructor accepts four parameters, which determine the bounding box x and y limits. You can also construct the BoundingBox using helper methods *BoundingBox.single_point()* and *BoundingBox.from_points()*

Class methods:

classmethod BoundingBox.**single_point**(*point*)
Creates a BoundingBox from a single point. The `point` parameter must be a *geometry.Vector* representing point coordinates. A single point BoundingBox has no width/height.

classmethod BoundingBox.**from_points**(*points*)
Creates a BoundingBox from points. The `points` must be a list of *geometry.Vector* instances, representing point coordinates.

If `points` list is empty, it will return bounding box with NaN values.

If `points` list has 1 point, it behaves exactly like *BoundingBox.single_point()*

If `points` list has 2 or more points, it will return smallest box which contains all provided points.

Instance properties:

BoundingBox.**min_x**
Gets min_x of the bounding box.

BoundingBox.**min_y**
Gets min_y of the bounding box.

BoundingBox.**max_x**
Gets max_x of the bounding box.

BoundingBox.**max_y**
Gets max_y of the bounding box.

BoundingBox.**is_nan**
Gets "not a number" status of the bounding box, as `bool`

BoundingBox.**center**
Gets the central point of the bounding box, as *geometry.Vector*.

BoundingBox.**dimensions**
Gets dimensions of the bounding box, as *geometry.Vector*, x being width and y being height.

Instance methods:

BoundingBox.**merge**(*other_bounding_box*)
Merges the bounding box with other and returns a new bounding box.

BoundingBox.**contains**(*other*)
Other can be *BoundingBox* or *geometry.Vector*. Returns `True` if bounding box contains other bounding box or point.

BoundingBox.**intersects**(*other_bounding_box*)
Returns `True` if bounding box intersects with other *geometry.BoundingBox*, otherwise returns `False`

BoundingBox.**intersection**(*other_bounding_box*)
If bounding box intersects with other *geometry.BoundingBox* a *BoundingBox* is returned which spans the intersection. If there's no intersection, an 'empty' *geometry.BoundingBox* is returned (all properties set to NaN)

BoundingBox.**grow**(*vector*)
Grows the bounding box by given vector (adds the vector's x and y value to the corresponding sides of the bounding box). The `vector` param must be *geometry.Vector*

2.6.8 `Alignment` reference

class geometry.**Alignment**

Enum type used to set Node's origin alignment to one of the 9 positions. See *nodes.Node.origin_alignment*

Available values are:

- `Alignment.none`
- `Alignment.top`
- `Alignment.bottom`
- `Alignment.left`
- `Alignment.right`
- `Alignment.top_left`
- `Alignment.bottom_left`
- `Alignment.top_right`
- `Alignment.bottom_right`
- `Alignment.center`

2.6.9 `PolygonType` reference

class geometry.**PolygonType**

Enum type returned by the *classify_polygon()* function. Available values:

- `PolygonType.convex_cw` - the list of points forms a convex polygon, the points are ordered clockwise
- `PolygonType.convex_ccw` - the list of points forms a convex polygon, the points are ordered counter clockwise
- `PolygonType.not_convex` - the list of points forms a non-convex polygon

2.6.10 `classify_polygon()` reference

geometry.**classify_polygon**(*polygon*)

Accepts a list of points (list of *geometry.Vector*) and returns if polygon formed by those points is convex or not. The function returns a *PolygonType* enum value.

```
from kaa.geometry import Vector, classify_polygon

print(classify_polygon([Vector(0, 0), Vector(10, 0), Vector(10, 10), Vector(0, 10)]))␣
↪ # PolygonType.conwex_ccw
print(classify_polygon([Vector(0, 0), Vector(0, 10), Vector(10, 10), Vector(10, 0)]))␣
↪ # PolygonType.conwex_cw
print(classify_polygon([Vector(0, 0), Vector(10, 0), Vector(2, 2), Vector(0, 10)]))
↪# PolygonType.not_convex
```

2.7 `input` — Handling input from keyboard, mouse and controllers

2.7.1 `InputManager` reference

class input.**InputManager**

Input manager object can be accessed via *Scene.input* property. It has three main features:

- Gives you access to specialized managers: *MouseManager*, *KeyboardManager*, *ControllerManager* and *SystemManager* - they offer methods to actively check for input from your code. For instance, you can ask the *KeyboardManager* if given key is pressed or released.
- Gives you access to a list of events which ocurred during the frame. This is achieved by calling the *InputManager.events()* method. Check out the *Event* documentation for a list of all available events that kaaengine detects.
- Allows you to subscribe to specific types of events by registering your own callback function. This is done using *InputManager.register_callback()* function.

And a number of other minor features.

Instance Properties:

InputManager.**keyboard**
: A get accessor returning *KeyboardManager* object which exposes methods to check for keyboard input. See the *KeyboardManager* documentation for a full list of available methods.

`InputManager.`**`mouse`**
: A get accessor returning *`MouseManager`* object which exposes methods to check for mouse input. See the *`MouseManager`* documentation for a full list of available methods.

`InputManager.`**`controller`**
: A get accessor returning *`ControllerManager`* object which exposes methods to check for controller input. See the *`ControllerManager`* documentation for a full list of available methods.

`InputManager.`**`system`**
: A get accessor returning *`SystemManager`* object which exposes methods to check for system input. See the *`SystemManager`* documentation for a full list of available methods.

`InputManager.`**`cursor_visible`**
: Gets or sets the visibility of the mouse cursor as `bool`.

Instance Methods:

`InputManager.`**`events`**`()`
: Returns a list of *`Event`* objects that ocurred during the last frame. Check out the *`Event`* instance documentation for details.

`InputManager.`**`register_callback`**(*event_type*, *callback_func*)
: Registers a callback function which will be called when specific event type(s) occur. Allows for an easy consumption of events you're interested in.

  The `event_type` parameter must be a specific *`Event`* subtype. You can also pass an iterable of those. Represents event type(s) you want to subscribe to.

  The `callback_func` must be a callable. It will get called each time given event type occurs, passing the event as parameter.

```
from kaa.input import EventType

def on_text_input(event):
    print('user typed this: {}'.format(event.keyboard_text.text))

def on_mouse_event(event):
    print('mouse button/wheel stuff happened!')

# somewhere inside a Scene instance...
self.input.register_callback(Event.keyboard_text, on_text_input)
self.input.register_callback([Event.mouse_button, Event.mouse_wheel], on_mouse_
↪event)
```

  Only one callback for given event type can be registered at a time. Registering another callback with the same event type will overwrite the previous one:

```
from kaa.input import EventType

def on_text_input_1(event):
    print('1 - user typed this: {}'.format(event.keyboard_text.text))

def on_text_input_2(event):
    print('2 - user typed this: {}'.format(event.keyboard_text.text))

# somewhere inside a Scene instance...
self.input.register_callback(Event.keyboard_text, on_text_input_1)
# this will cancel the previous callback (i.e. on_text_input_1 will never be␣
↪called):
self.input.register_callback(Event.keyboard_text, on_text_input_2)
```

If you pass `None` as callback_func, it will unregister the currently existing callback for that even type or do nothing if no callback for that type is currently registered.

```
from kaa.input import EventType

def on_text_input(event):
    print('user typed this: {}'.format(event.keyboard_text.text))

# somewhere inside a Scene instance...
self.input.register_callback(Event.keyboard_text, on_text_input)
self.input.register_callback(Event.keyboard_text, None) # unregisters the␣
↪callback, on_text_input won't be called
```

2.7.2 KeyboardManager reference

class input.**KeyboardManager**

Keyboard manager can be accessed via the *InputManager.keyboard* property.

It allows to check the state (pressed or released) of given key.

Instance methods:

KeyboardManager.**is_pressed**(*keycode*)

Checks if a specific key is pressed - keycode param must be a *Keycode* enum value.

```
from kaa.input import Keycode

# somewhere inside a Scene instance...
if self.input.keyboard.is_pressed(Keycode.w):
    # ... do something if w is pressed
if self.input.keyboard.is_pressed(Keycode.W):
    # ... do something if W is pressed
if self.input.keyboard.is_pressed(Keycode.return_):
    # ... do something if ENTER key is pressed
```

KeyboardManager.**is_released**(*keycode*)

Checks if a specific key is released - keycode param must be a *Keycode* enum value.

```
from kaa.input import Keycode

# somewhere inside a Scene instance...
if self.input.keyboard.is_released(Keycode.w):
    # ... do something if w is released
if self.input.keyboard.is_released(Keycode.W):
    # ... do something if W is released
if self.input.keyboard.is_released(Keycode.return_):
    # ... do something if ENTER key is released
```

2.7.3 MouseManager reference

class input.**MouseManager**

Mouse manager can be accessed via the *InputManager.mouse* property.

The manager allows to check for the mouse buttons state (pressed/released). It also allows to get the mouse pointer position.

Instance properties:

`MouseManager.`**`relative_mode`**
: Gets or sets relative mode (as bool). Default is `False`. Enabling relative mode has two effects: it hides the mouse pointer and it makes mouse motion events (*`MouseMotionEvent`*) be published all the time (by default those events are published only if mouse moves within game's window). Disabling the relative mode shows the mouse pointer and makes mouse motion events be published only if mouse movement occurs within the window.

`MouseManager.`**`cursor_visible`**
: Gets or sets whether the system cursor pointer should be visible.

Instance methods:

`MouseManager.`**`is_pressed`**`(`*mousebutton*`)`
: Checks if given mouse button is pressed - mousebutton param must be a *`MouseButton`* enum value.

```
from kaa.input import MouseButton

#somewhere inside a Scene instance...
if self.input.mouse.is_pressed(MouseButton.left):
    # do something if the left mouse button is pressed
```

`MouseManager.`**`is_released`**`(`*mousebutton*`)`
: Checks if given mouse button is released - mousebutton param must be a *`MouseButton`* enum value.

```
from kaa.input import MouseButton

#somewhere inside a Scene instance...
if self.input.mouse.is_released(MouseButton.middle):
    # do something if the middle mouse button is released
```

`MouseManager.`**`get_position`**`()`
: Returns current mouse pointer position as a *`geometry.Vector`*.

  IMPORTANT: the position is in the display screen frame of reference using the *virtual resolution coordinate system*). It is NOT a position of a mouse cursor on the scene. To convert the position to the scene frame of reference use the *`engine.Camera.unproject_position()`* method.

```
#somewhere inside a Scene instance...
pos = self.input.mouse.get_position():
print(pos)  # V[145.234, 345.343]
```

2.7.4 `ControllerManager` reference

`class` `input.`**`ControllerManager`**

Controller Manager can be accessed via the *InputManager.controller* property.

The manager exposes methods for checking the state of controller's buttons, sticks and triggers. All major controller types are supported.

Unlike mouse or keyboard, multiple controllers can be connected and used simultaneously, therefore all manager methods require passing a controller ID.

You can get the controller ID when controller is first connected. Kaa engine will publish a `ControllerEvent` having `connected` flag set to `True`. An `id` field on the event is the controller ID you're looking for.

When a controller disconnects, you will receive a `ControllerEvent` with `connected` flag set to `True`.

Your game code should always keep track of all currently connected controllers (their IDs).

Below is a basic example of keeping track of connected controller IDs and checking few selected properties of each connected controller:

```
from kaa.engine import Engine, Scene
from kaa.geometry import Vector
from kaa.input import Keycode, ControllerButton, ControllerAxis

class MyScene(Scene):

    def __init__(self):
        self.connected_controller_ids = []
        self.frame_count = 0

    def update(self, dt):
        self.frame_count += 1
        for event in self.input.events():

            if event.controller_device:
                if event.controller_device.is_added:
                    print('New controller connected: id is {}'.format(event.
↪controller_device.id))
                    self.connected_controller_ids.append(event.controller_device.id)
                elif event.controller_device.is_removed:
                    print('Controller disconnected: id is {}'.format(event.controller_
↪device.id))
                    self.connected_controller_ids.remove(event.controller_device.id)

            if event.system and event.system.quit:
                self.engine.quit()

        # Check a few properties of each connected controller:
        for controller_id in self.connected_controller_ids:
            a_button_pressed = self.input.controller.is_pressed(ControllerButton.a,␣
↪controller_id)
            b_button_pressed = self.input.controller.is_pressed(ControllerButton.b,␣
↪controller_id)
            left_stick_x = self.input.controller.get_axis_motion(ControllerAxis.left_
↪x, controller_id)
            left_stick_y = self.input.controller.get_axis_motion(ControllerAxis.left_
↪y, controller_id)
            print('Controller {}. A pressed:{}, B pressed:{}, left stick pos: {},{}'.
↪format(controller_id,
                a_button_pressed, b_button_pressed, left_stick_x, left_stick_y))


with Engine(virtual_resolution=Vector(400, 200)) as engine:
    scene = MyScene()
    engine.window.size = Vector(400, 200)
    engine.window.center()
    engine.run(scene)
```

Instance methods

`ControllerManager.`**`is_connected`**(*controller_id*)
Checks connection status of a given controller_id.

`ControllerManager.`**`is_pressed`**(*controller_button*, *controller_id*)
Checks if given controller button is pressed. The controller_button param must be a *ControllerButton*

enum value. Check out the *example above* on how to obtain the controller_id.

For example, to check the state of B button on controller 0:

```
from kaa.input import ControllerButton

# somewhere in the Scene class:
if self.input.controller.is_pressed(ControllerButton.b, 0):
    print('B is pressed on controller 0!')
```

ControllerManager.**is_released**(*controller_button*, *controller_id*)

Checks if given controller button is released on given controller. The controller_button param must be a *ControllerButton* enum value. Check out the *example above* on how to obtain the controller_id.

For example, to check the state of B button on controller 2:

```
from kaa.input import ControllerButton

# somewhere in the Scene class:
if self.input.controller.is_released(ControllerButton.b, 2):
    print('B is released on controller 2!')
```

ControllerManager.**is_axis_pressed**(*axis*, *controller_id*)

Checks if given stick axes or trigger is in non-zero position. The axis param must be of *ControllerAxis* enum value. Check out the *example above* on how to obtain the controller_id.

For example, to check if controller 1 left trigger is pressed:

```
from kaa.input import ControllerAxis

# somewhere in the Scene class:
if self.input.controller.is_axis_pressed(ControllerAxis.trigger_left, 1):
    print('Left trigger is pressed!')
```

ControllerManager.**is_axis_released**(*axis*, *controller_id*)

Same as above, but checks if given stick axes or trigger is in a zero position. The axis param must be of *ControllerAxis* enum value. Check out the *example above* on how to obtain the controller_id.

ControllerManager.**get_axis_motion**(*axis*, *controller_id*)

Gets an exact value of given stick axes motion or trigger as a number between 0 (stick axes or trigger in zero position) and 1 (stick axes or trigger in max position). The axis param must be of *ControllerAxis* enum value. Check out the *example above* on how to obtain the controller_id.

For example, to check the state of controller 0 left trigger:

```
from kaa.input import ControllerAxis

# somewhere in the Scene class:
val = self.input.controller.get_axis_motion(ControllerAxis.trigger_right, 0):
print('Controller 0, pulling left trigger {} percent :)'.format(val*100))
```

ControllerManager.**get_name**(*controller_id*)

Returns a name of a controller. Check out the *example above* on how to obtain the controller_id.

ControllerManager.**get_triggers**(*controller_id*)

Returns state of both triggers in a single *geometry.Vector* object. Vector's x value is left trigger and vector's y is right trigger. Check out the *example above* on how to obtain the controller_id.

The values returned are between 0 (trigger is fully released) to 1 (trigger is fully pressed)

ControllerManager.**get_sticks**(*compound_axis*, *controller_id*)
Returns state of given stick as a *geometry.Vector*.

The compound_axis parameter must be of *CompoundControllerAxis* enum value.

Check out the *example above* on how to obtain the controller_id.

For example, to get the controller 1 left stick position:

```
# somewhere in the Scene class:
val = self.input.controller.get_axis_motion(CompoundControllerAxis.left_stick, 1):
print('Controller 1, left stick position is {}'.format(val))
```

2.7.5 SystemManager reference

class input.**SystemManager**

System Manager can be accessed via the *InputManager.system* property.

The manager exposes methods for working with system related input such as clipboard.

Instance methods:

SystemManager.**get_clipboard_text**()
Gets text from the system clipboard

SystemManager.**set_clipboard_text**(*text*)
Puts the text in the system clipboard

2.7.6 Event reference

class input.**Event**

As the game is running, a lot of things happen: the player may press or release keyboard keys or mouse buttons, interact with controller, he can also interact with the window in which your game is running, e.g. maximize or minimize it, and so on. Kaa engine detects all those events and makes them consumable either via *InputManager.events()* method or by registering a callback function *InputManager.register_callback()*.

Each *Event* **instance** has identical structure with the following instance properties:

- type - returns event type
- timestamp - returns time of the event occurrence
- system - stores *SystemEvent* instance if this event is a system related event, otherwise it will be None
- window - stores *WindowEvent* instance if this event is a window related event, otherwise it will be None
- keyboard_key - stores *KeyboardKeyEvent* instance if this event is a keyboard key related event, otherwise it will be None
- keyboard_text - stores *KeyboardTextEvent* instance if this event is a keyboard text related event, otherwise it will be None
- mouse_button - stores *MouseButtonEvent* instance if this event is a mouse button related event, otherwise it will be None
- mouse_motion - stores *MouseMotionEvent* instance if this event is a mouse motion related event, otherwise it will be None
- mouse_wheel - stores *MouseWheelEvent* instance if this event is a mouse wheel related event, otherwise it will be None

- `controller_device` - stores *ControllerDeviceEvent* instance if this event is a controller device related event, otherwise it will be `None`
- `controller_button` - stores *ControllerButtonEvent* instance if this event is a controller button related event, otherwise it will be `None`
- `controller_axis` - stores *ControllerAxisEvent* instance if this event is a controller axis related event, otherwise it will be `None`
- `audio` - stores *AudioEvent* instance if this event is an audio related event, otherwise it will be `None`

Depending on the type of the event only one property will be non-null while all the other properties will be null. This design usually results in a following way of handling events in the code:

```
# ... inside a Scene...
def update(self, dt):

    for event in self.input.events():
        if event.system:
            # do something if it's a system event
        elif event.window:
            # do something if it's a window event
        elif event.keyboard_key:
            # do something if it's a keyboard key event
        elif event.keyboard_text:
            # do something if it's a keyboard text event
        elif event.mouse_button:
            # do something if it's a mouse button event
        elif event.controller_button:
            # do something if it's a controller button event
        elif event.audio:
            # do something if it's audio event
        # ... and so on ...
```

Event **class** also has descriptors ("static properties") that return appropriate event types:

- Event.system - returns *SystemEvent* type
- Event.window - returns *WindowEvent* type
- Event.keyboard_key - returns *KeyboardKeyEvent* type
- Event.keyboard_text - returns *KeyboardTextEvent* type
- Event.mouse_button - returns *MouseButtonEvent* type
- Event.mouse_motion - returns *MouseMotionEvent* type
- Event.mouse_wheel - returns *MouseWheelEvent* type
- Event.controller_device - returns *ControllerDeviceEvent* type
- Event.controller_button - returns *ControllerButtonEvent* type
- Event.controller_axis - returns *ControllerAxisEvent* type
- Event.audio - returns *AudioEvent* type

which allows checking the `type` property on the event instance:

```
# ... inside a Scene...
def update(self, dt):
```

(continues on next page)

(continued from previous page)

```
    for event in self.input.events():
        if event.type == Event.system:
            # do something
        elif event.type == Event.keyboard_key:
            # do something ...
        elif event.type == Event.controller_axis:
            # do something ...
        # ... and so on
```

2.7.7 `KeyboardKeyEvent` reference

class input.**KeyboardKeyEvent**

Represents an event of pressing or releasing a keyboard key.

See also: *KeyboardTextEvent*

Instance properties:

KeyboardEvent.**key**
: Returns the key this event is referring to, as *Keycode* enum value.

KeyboardEvent.**is_key_down**
: Returns `True` if the key was pressed.

KeyboardEvent.**is_key_up**
: Returns `True` if the key was released.

2.7.8 `KeyboardTextEvent` reference

class input.**KeyboardTextEvent**

Represents an event of text being produced by the keyboard buffer. It lets you conveniently work with the text being typed in by the player.

Instance properties:

KeyboardTextEvent.**text**
: Returns string with the text typed in.

  For example, imagine a user with a polish keyboard pressing shift key, right alt and 's' keys, holding it for some time and then releasing all pressed keys.

  In a text editor it would result in typing something like this:

  ŚŚŚŚŚŚ

  The way ths will be represented in the kaaengine event flow:

  1) You will first receive three *KeyboardKeyEvent* events: one for pressing the shift key, another for pressing the alt key and one for pressing the s key
  2) You will then receive a number of *KeyboardTextEvent* events, in this case we have six 'Ś' characters typed, so you will get six events. Reading the `text` property on *KeyboardTextEvent* will return "Ś" string.
  3) Finally, you will first receive three *KeyboardKeyEvent* events: one for releasing the shift key, another for releasing the alt key and another one for releasing the s key

2.7.9 `MouseButtonEvent` reference

class input.**MouseButtonEvent**

Represents a mouse button related event, such as pressing or releasing a mouse button.

```
# ... inside a Scene instance...
for event in self.input.events():
    if event.mouse_button:
        if event.mouse_button.is_button_down:
            print("Mouse button {} is DOWN. Mouse position: {}.".
↪format(event.mouse_button.button,
                  event.mouse_button.position))
        elif event.mouse_button.is_button_up:
            print("Mouse button {} is UP. Mouse position: {}.".format(event.
↪mouse_button.button,
                  event.mouse_button.position))
```

Instance properties:

MouseButtonEvent.**button**
: Returns the button this event is referring to, as *MouseButton* enum value.

MouseButtonEvent.**is_button_down**
: Returns `True` if the button was pressed.

MouseButtonEvent.**is_button_up**
: Returns `True` if the button was released.

MouseButtonEvent.**position**
: Returns mouse pointer position, at the moment of the click, as *geometry.Vector*. *IMPORTANT*: the position is in the display screen frame of reference (using the *virtual resolution coordinate system*). It is NOT a position of a mouse cursor on the scene. To convert the position to the scene frame of reference use the *engine.Camera.unproject_position()* method on the Camera object.

```
# somewhere inside the Scene instance:
for event in self.input.events():
    if event.mouse_button:
        if event.mouse_button.is_button_down and event.mouse_button.button ==␣
↪MouseButton.left:
            mouse_pos_absolute = event.mouse_button.position
            mouse_pos_on_scene = self.camera.unproject_position(mouse_pos_
↪absolute)
```

2.7.10 `MouseMotionEvent` reference

class input.**MouseMotionEvent**

Represents a mouse motion event (changing mouse pointer position). By default those events are published when mouse pointer is within the window. You can enable the `relative_mode` on the *MouseManager* - it hides the mouse pointer and makes mouse motion events be published whenever the pointer is moved (inside or outside of the window).

```
# ... inside a Scene instance...
for event in self.input.events():
    if event.mouse_motion:
         print("Mouse motion detected! New position is: {}.".format(event.
↪mouse_motion.position))
```

Instance properties:

`MouseMotionEvent.`**`position`**
: Returns mouse pointer position as *`geometry.Vector`*. *IMPORTANT*: the position is in the display screen frame of reference (using the *virtual resolution coordinate system*). It is NOT a position of a mouse cursor on the scene. To convert the position to the scene frame of reference use the *`engine.Camera.unproject_position()`* method on the Camera object.

```
# somewhere inside the Scene instance:
for event in self.input.events():
    if event.mouse_motion:
        mouse_pos_absolute = event.mouse_button.position
        mouse_pos_on_scene = self.camera.unproject_position(mouse_pos_absolute)
```

`MouseMotionEvent.`**`motion`**
: Returns mouse pointer motion (difference between the current and previous position) as *`geometry.Vector`*.

2.7.11 `MouseWheelEvent` reference

`class` `input.`**`MouseWheelEvent`**

Represents a mouse wheel related event.

Instance properties:

`MouseWheelEvent.`**`scroll`**
: Returns a *`geometry.Vector`* indicating whether the mouse wheel was scrolled up or down. The `y` property in the returned vector holds the value, the `x` will always be zero.

```
# ... inside a Scene instance...
for event in self.input.events():
    if event.mouse_wheel:
        print("Mouse wheel event detected. Scroll is: {}.".format(event.mouse_
↪wheel.scroll))
```

2.7.12 `ControllerDeviceEvent` reference

Represents a controller device related event, such as controller connected or disconnected.

`class` `input.`**`ControllerDeviceEvent`**

```
# ... inside a Scene instance...
for event in self.input.events():
    if event.controller_device:
        if event.controller_device.is_added:
            print("Controller with id={} connected.".format(event.controller_
↪device.id)
        elif event.controller_device.is_removed:
            print("Controller with id={} disconnected.".format(event.controller_
↪device.id)
```

Instance properties:

`ControllerDeviceEvent.`**`id`**
: Returns an id of controller this event is referring to.

`ControllerDeviceEvent.`**`is_added`**
Returns `True` if controller was connected.

`ControllerDeviceEvent.`**`is_removed`**
Returns `True` if controller was disconnected.

2.7.13 `ControllerButtonEvent` reference

Represents a controller button related event, such as controller button pressed or released.

`class` `input.`**`ControllerButtonEvent`**
Note: Controller triggers are considered sticks (axis) not buttons! Use *`ControllerAxisEvent`* to check out events representing triggers changing state.

```
# ... inside a Scene instance...
for event in self.input.events():
    if event.controller_button:
        if event.controller_button.is_button_down:
            print("Controller button {} on controller id={} was pressed.".format(
                  event.controller_button.button, event.controller_button.id)
        elif event.controller_button.is_button_up:
            print("Controller button {} on controller id={} was released.".format(
                  event.controller_button.button, event.controller_button.id)
```

Instance properties:

`ControllerButtonEvent.`**`id`**
Returns an id of controller this event is referring to.

`ControllerButtonEvent.`**`button`**
Returns controller button this event is referring to, as *`ControllerButton`* enum value.

`ControllerButtonEvent.`**`is_button_down`**
Returns `True` if the button was pressed.

`ControllerButtonEvent.`**`is_button_up`**
Returns `True` if the button was released.

2.7.14 `ControllerAxisEvent` reference

Represents a controller axis related event, such as controller stick or trigger state change.

```
# ... inside a Scene instance...
for event in self.input.events():
    if event.controller_axis:
        print("Controller axis {} on controller id={} changed its state. New␣
↪state is {}.".format(
              event.controller_axis.axis, event.controller_axis.id, event.
↪controller_axis.motion)
```

`class` `input.`**`ControllerAxisEvent`**

Instance properties

`ControllerAxisEvent.`**`id`**
Returns an id of controller this event is referring to.

`ControllerAxisEvent.`**`axis`**
Returns axis (controller stick or trigger) this event is referring to, as *`ControllerAxis`* enum value.

ControllerAxisEvent.**motion**
: Returns the axis (controller stick or trigger) state, as a *geometry.Vector*. The length of the vector will be between 0 (stick or trigger is in neutral position) and 1 (stick or trigger is in its maximum position)

2.7.15 AudioEvent reference

class input.**AudioEvent**

Represents an audio related event.

Instance properties:

AudioEvent.**music_finished**
: Returns True if current music track finished playing.

2.7.16 WindowEvent reference

class input.**WindowEvent**

Represents a window related event.

Instance properties:

WindowEvent.**is_shown**
: Returns True if the window was shown.

WindowEvent.**is_exposed**
: Returns True if the window was exposed.

WindowEvent.**is_moved**
: Returns True if the window was moved.

WindowEvent.**is_resized**
: Returns True if the window was resized.

WindowEvent.**is_minimized**
: Returns True if the window was minimized.

WindowEvent.**is_maximized**
: Returns True if the window was maximized.

WindowEvent.**is_restored**
: Returns True if the window was restored.

WindowEvent.**is_enter**
: Returns True if the mouse pointer entered the window area.

WindowEvent.**is_leave**
: Returns True if the mouse pointer left the window area.

WindowEvent.**is_focus_gained**
: Returns True if the window gained a focus.

WindowEvent.**is_focus_lost**
: Returns True if the window lost a focus.

WindowEvent.**is_close**
: Returns True if the window was closed.

2.7.17 `SystemEvent` reference

class input.**SystemEvent**

Represents a system related event.

Instance properties:

SystemEvent.**quit**
: Returns `True` if the game proces is terminating.

SystemEvent.**clipboard_updated**
: Returns `True` if the system clipboard was updated. You may call *SystemManager.get_clipboard_text()* method to check out the text in the system clipboard.

2.7.18 `Keycode` reference

class input.**Keycode**

Enum type for referencing keyboard keys when working with *KeyboardManager* and *KeyboardKeyEvent*.

Available values are:

- `Keycode.unknown`
- `Keycode.return_`
- `Keycode.escape`
- `Keycode.backspace`
- `Keycode.tab`
- `Keycode.space`
- `Keycode.exclaim`
- `Keycode.quotedbl`
- `Keycode.hash`
- `Keycode.percent`
- `Keycode.dollar`
- `Keycode.ampersand`
- `Keycode.quote`
- `Keycode.leftparen`
- `Keycode.rightparen`
- `Keycode.asterisk`
- `Keycode.plus`
- `Keycode.comma`
- `Keycode.minus`
- `Keycode.period`
- `Keycode.slash`
- `Keycode.num_0`

- Keycode.num_1
- Keycode.num_2
- Keycode.num_3
- Keycode.num_4
- Keycode.num_5
- Keycode.num_6
- Keycode.num_7
- Keycode.num_8
- Keycode.num_9
- Keycode.colon
- Keycode.semicolon
- Keycode.less
- Keycode.equals
- Keycode.greater
- Keycode.question
- Keycode.at
- Keycode.leftbracket
- Keycode.backslash
- Keycode.rightbracket
- Keycode.caret
- Keycode.underscore
- Keycode.backquote
- Keycode.a
- Keycode.b
- Keycode.c
- Keycode.d
- Keycode.e
- Keycode.f
- Keycode.g
- Keycode.h
- Keycode.i
- Keycode.j
- Keycode.k
- Keycode.l
- Keycode.m
- Keycode.n

- Keycode.o
- Keycode.p
- Keycode.q
- Keycode.r
- Keycode.s
- Keycode.t
- Keycode.u
- Keycode.v
- Keycode.w
- Keycode.x
- Keycode.y
- Keycode.z
- Keycode.A
- Keycode.B
- Keycode.C
- Keycode.D
- Keycode.E
- Keycode.F
- Keycode.G
- Keycode.H
- Keycode.I
- Keycode.J
- Keycode.K
- Keycode.L
- Keycode.M
- Keycode.N
- Keycode.O
- Keycode.P
- Keycode.Q
- Keycode.R
- Keycode.S
- Keycode.T
- Keycode.U
- Keycode.V
- Keycode.W
- Keycode.X

- Keycode.Y
- Keycode.Z
- Keycode.capslock
- Keycode.F1
- Keycode.F2
- Keycode.F3
- Keycode.F4
- Keycode.F5
- Keycode.F6
- Keycode.F7
- Keycode.F8
- Keycode.F9
- Keycode.F10
- Keycode.F11
- Keycode.F12
- Keycode.printscreen
- Keycode.scrolllock
- Keycode.pause
- Keycode.insert
- Keycode.home
- Keycode.pageup
- Keycode.delete
- Keycode.end
- Keycode.pagedown
- Keycode.right
- Keycode.left
- Keycode.down
- Keycode.up
- Keycode.numlockclear
- Keycode.kp_divide
- Keycode.kp_multiply
- Keycode.kp_minus
- Keycode.kp_plus
- Keycode.kp_enter
- Keycode.kp_1
- Keycode.kp_2

- Keycode.kp_3
- Keycode.kp_4
- Keycode.kp_5
- Keycode.kp_6
- Keycode.kp_7
- Keycode.kp_8
- Keycode.kp_9
- Keycode.kp_0
- Keycode.kp_period
- Keycode.application
- Keycode.power
- Keycode.kp_equals
- Keycode.F13
- Keycode.F14
- Keycode.F15
- Keycode.F16
- Keycode.F17
- Keycode.F18
- Keycode.F19
- Keycode.F20
- Keycode.F21
- Keycode.F22
- Keycode.F23
- Keycode.F24
- Keycode.execute
- Keycode.help
- Keycode.menu
- Keycode.select
- Keycode.stop
- Keycode.again
- Keycode.undo
- Keycode.cut
- Keycode.copy
- Keycode.paste
- Keycode.find
- Keycode.mute

- Keycode.volumeup
- Keycode.volumedown
- Keycode.kp_comma
- Keycode.kp_equalsas400
- Keycode.alterase
- Keycode.sysreq
- Keycode.cancel
- Keycode.clear
- Keycode.prior
- Keycode.return2
- Keycode.separator
- Keycode.out
- Keycode.oper
- Keycode.clearagain
- Keycode.crsel
- Keycode.exsel
- Keycode.kp_00
- Keycode.kp_000
- Keycode.thousandsseparator
- Keycode.decimalseparator
- Keycode.currencyunit
- Keycode.currencysubunit
- Keycode.kp_leftparen
- Keycode.kp_rightparen
- Keycode.kp_leftbrace
- Keycode.kp_rightbrace
- Keycode.kp_tab
- Keycode.kp_backspace
- Keycode.kp_a
- Keycode.kp_b
- Keycode.kp_c
- Keycode.kp_d
- Keycode.kp_e
- Keycode.kp_f
- Keycode.kp_xor
- Keycode.kp_power

- Keycode.kp_percent
- Keycode.kp_less
- Keycode.kp_greater
- Keycode.kp_ampersand
- Keycode.kp_dblampersand
- Keycode.kp_verticalbar
- Keycode.kp_dblverticalbar
- Keycode.kp_colon
- Keycode.kp_hash
- Keycode.kp_space
- Keycode.kp_at
- Keycode.kp_exclam
- Keycode.kp_memstore
- Keycode.kp_memrecall
- Keycode.kp_memclear
- Keycode.kp_memadd
- Keycode.kp_memsubtract
- Keycode.kp_memmultiply
- Keycode.kp_memdivide
- Keycode.kp_plusminus
- Keycode.kp_clear
- Keycode.kp_clearentry
- Keycode.kp_binary
- Keycode.kp_octal
- Keycode.kp_decimal
- Keycode.kp_hexadecimal
- Keycode.lctrl
- Keycode.lshift
- Keycode.lalt
- Keycode.lgui
- Keycode.rctrl
- Keycode.rshift
- Keycode.ralt
- Keycode.rgui
- Keycode.mode
- Keycode.audionext

- Keycode.audioprev
- Keycode.audiostop
- Keycode.audioplay
- Keycode.audiomute
- Keycode.mediaselect
- Keycode.www
- Keycode.mail
- Keycode.calculator
- Keycode.computer
- Keycode.ac_search
- Keycode.ac_home
- Keycode.ac_back
- Keycode.ac_forward
- Keycode.ac_stop
- Keycode.ac_refresh
- Keycode.ac_bookmarks
- Keycode.brightnessdown
- Keycode.brightnessup
- Keycode.displayswitch
- Keycode.kbdillumtoggle
- Keycode.kbdillumdown
- Keycode.kbdillumup
- Keycode.eject
- Keycode.sleep

2.7.19 MouseButton reference

class input.**MouseButton**

Enum type for referencing mouse buttons when working with *MouseManager* and *MouseButtonEvent*.

Available values are:

- MouseButton.left
- MouseButton.middle
- MouseButton.right
- MouseButton.x1
- MouseButton.x2

2.7.20 `ControllerButton` reference

class input.**ControllerButton**

Enum type for referencing controller buttons when working with *ControllerManager* and *ControllerButtonEvent*. Note that left and right triggers are not buttons, they're considered axis (see *ControllerAxisEvent*)

Available values are:

- `ControllerButton.a`
- `ControllerButton.b`
- `ControllerButton.x`
- `ControllerButton.y`
- `ControllerButton.back`
- `ControllerButton.guide`
- `ControllerButton.start`
- `ControllerButton.left_stick`
- `ControllerButton.right_stick`
- `ControllerButton.left_shoulder`
- `ControllerButton.right_shoulder`
- `ControllerButton.dpad_up`
- `ControllerButton.dpad_down`
- `ControllerButton.dpad_left`
- `ControllerButton.dpad_right`

2.7.21 `ControllerAxis` reference

class input.**ControllerAxis**

Enum type for referencing controller axes when working with *ControllerManager* and *ControllerAxisEvent*.

Available values are:

- `ControllerAxis.left_y`
- `ControllerAxis.left_x`
- `ControllerAxis.right_x`
- `ControllerAxis.right_y`
- `ControllerAxis.trigger_left`
- `ControllerAxis.trigger_right`

2.7.22 `CompoundControllerAxis` reference

class input.**CompoundControllerAxis**

Enum type for referencing sticks (left or right) when working with some of *ControllerManager* methods.

Available values are:

- `CompoundControllerAxis.left_stick`
- `CompoundControllerAxis.right_stick`

2.8 `log` — kaaengine logging settings

By default kaa logs everything to stderr.

2.8.1 `get_core_logging_level()` reference

log.**get_core_logging_level**(*core_catgory*)

Gets logging level for given category. The `core_category` param must be a `CoreLogCategory` enum value.

2.8.2 `set_core_logging_level()` reference

log.**set_core_logging_level**(*core_catgory*, *level*)

Sets a logging level for given category.

The `core_category` param must be a string with kaa module name.

List of available kaa module names: “nodes”, “node_ptr”, “engine”, “files”, “log”, “renderer”, “images”, “input”, “audio”, “scenes”, “shapes”, “physics”, “resources”, “resources_manager”, “sprites”, “window”, “geometry”, “fonts”, “timers”, “transitions”, “node_transitions”, “camera”, “views”, “spatial_index”, “threading”, “utils”, “embedded_data”, “easings”, “shaders”, “other”, “app”, “wrapper”

The `level` parameter must be a *CoreLogLevel* enum value.

Note: By default kaa logs everything to stderr.

```
from kaa.log import set_core_logging_level, CoreLogCategory, CoreLogLevel

set_core_logging_level("audio", CoreLogLevel.verbose)
```

2.8.3 `CoreLogLevel` reference

class log.**CoreLogLevel**

Enum type used to reference log levels. Available values are:

- `CoreLogLevel.verbose`
- `CoreLogLevel.debug`
- `CoreLogLevel.info`
- `CoreLogLevel.warn`

- `CoreLogLevel.error`
- `CoreLogLevel.critical`

2.9 `nodes` — Your objects on the scene

2.9.1 `Node` reference

Nodes are the core concept of the kaa engine. They're "objects" which you can add to the Scene. Each Node has its spatial properties such as position, rotation or scale. A Node may also have a sprite (graphics loaded from a file), *which can be animated*. Nodes have other properties such as z_index, shape, color, origin etc. All those properties are described in the documentation below.

A Node can have child Nodes, which you can add with the *`Node.add_child()`* method, thus creating a tree-like structure of nodes on the scene. As the parent node gets transformed (changes its position, rotation or scale) all its children nodes will transform accordingly.

Each *`engine.Scene`* instance has a root node - this is the first node on the Scene to which you can start adding your own Nodes.

A node without a sprite image (or without a shape and color properties set explicitly) will be just a logical entity on the scene, in other words: you won't see it. Such logical Nodes are often very useful, for example, as a containers for grouping other nodes.

Although the bare *`Node`* will do its job well and allow you to create simple games, the Kaa engine comes with a collection of other specialized Nodes (they all inherit from the *`Node`* class):

- *`physics.SpaceNode`* - a container node to simulate the physical environment.
- *`physics.BodyNode`* - a physical node which can have hitbox nodes. Can interact with other BodyNodes. Must be a direct child of SpaceNode. Can have zero or more Hitbox Nodes.
- *`physics.HitboxNode`* - defines an area that can collide with other hitboxes, and allows wiring up your own collision handler function. Hitbox Node must be a child node of a BodyNode.
- *`fonts.TextNode`* - a node used to render text on the screen.

For your game's actual objects such as Player, Enemy, Bullet, etc. we recommend writing classes that inherit from the Node class (or BodyNode if you want the object to utilize *kaaengine's physics features*).

class nodes.**Node**(*position=Vector(0, 0)*, *rotation=0*, *scale=Vector(1, 1)*, *z_index=None*, *color=Color(0, 0, 0, 0)*, *sprite=None*, *shape=None*, *origin_alignment=Alignment.center*, *lifetime=None*, *transition=None*, *visible=True*, *views=None*)

A basic example how to create a new Node (with a sprite) and add it to the Scene:

```
from kaa.nodes import Node
from kaa.sprites import Sprite
from kaa.geometry import Vector
import os

# inside a Scene's __init__ :
my_sprite = Sprite(os.path.join('assets', 'gfx', 'arrow.png'))  # create a sprite␣
↪from image file
self.node = Node(position=Vector(100, 100), sprite=my_sprite))  # create a Node␣
↪at (100, 100) with the sprite
self.root.add_child(self.node)  # until you add the Node to the Scene it won't␣
↪not show up on the screen!
```

Instance Properties:

Node.**children**
Returns a list of child nodes of this Node.

Node.**scene**
Returns a Scene instance to which this Node belongs. Will be None if the node has not been added to any Scene yet. Use *Node.add_child()* method to add nodes. Each Scene has a root node to which you can add nodes.

Node.**position**
Gets or sets node position, as a *geometry.Vector*.

IMPORTANT: Node position is always get or set relative to its parent node. To get the absolute position, use the *absolute_position* property.

If the Node is few levels deep in the nodes hierarchy and you want to know the position of the node in relation to one of its ancestors, use *get_relative_position()* method.

```
from kaa.nodes import Node
from kaa.geometry import Vector

# inside a Scene's __init__ :
self.node1 = Node(position = Vector(100, 100))
self.root.add_child(self.node1)  # adding to scene's root node, so node1 absolute␣
↪position is (100, 100)
# create a child node
self.node2 = Node(position = Vector(-20, 30))
self.node1.add_child(self.node2)
print(self.node2.position) # prints out V[-20, 30]
print(self.node2.absolute_position) # prints out V[80, 130]
```

Also see: *Node origin points*.

Node.**absolute_position**
Read only. Gets an absolute position of the node, i.e. the position on the scene. Returns *geometry.Vector*.

Check out the example in the *position* property section.

Node.**parent**
Retruns this node's parent *Node*, or None in case of the root node.

Node.**root_distance**
Retruns this node's distance to the root node (counted as number of parents on the way to the root), or None in case of the root node.

Node.**z_index**
Gets or sets node z_index (integer). Nodes with higher z_index will overlap those with lower z_index when drawn on the screen. Default z_index is None meaning the node will inherit z_index value from its parent. To find the actual z_index value in this case, use *effective_z_index property*.

Scene's root node (*engine.Scene.root*) has z_index = 0.

Note: If parent and child nodes have the same z_index, then the child node will be rendered on top of the parent.

Node.**effective_z_index**
Gets effective z_index value of the node. Use it to find actual the actual z_index value when node inherits it from its parent.

```
# ... inside a Scene class...

node = Node(z_index=15)
child = Node()  # z_index is None

node.add_child(child)
self.root.add_child(node)

print(child.z_index)  # prints None
print(child.effective_z_index)  # prints 15
```

Node.**rotation**

Gets or sets node rotation, in radians. There is no capping value, meaning you can set it to values greater than `math.pi*2` or lower than `-math.pi*2`.

IMPORTANT: Node rotation is always get or set relative to its parent node. To get the absolute rotation, use the *absolute_rotation* property.

Changing node rotation will make the node rotate around its origin point. Read more about *Node origin points*.

```
import math
from kaa.nodes import Node
from kaa.geometry import Vector

# inside a Scene's __init__ :
# add node 1
self.node1 = Node(position = Vector(100, 100), rotation=math.pi / 4)
self.root.add_child(self.node1)
# add node 2 as child of node 1
self.node2 = Node(position = Vector(10, 10), rotation=math.pi / 4)
self.node1.add_child(self.node2)

print(self.node1.rotation) # 0.7853981633974483 (math.pi / 4)
print(self.node2.rotation) # 0.7853981633974483 (math.pi / 4)
print(self.node2.absolute_rotation) # 1.5707963705062866 (math.pi / 2)
```

Node.**absolute_rotation**

Read only. Returns an absolute rotation of the node, in radians. Check out the example in the *rotation* property section.

Node.**rotation_degrees**

Same as *rotation* property, but uses degrees (as float). There is no capping value, meaning you can set it to values greater than 360 degrees or smaller than -360 degrees.

Changing node rotation will make the node rotate around its origin point. Read more about *Node origin points*.

See also: *absolute_rotation_degrees*.

Node.**absolute_rotation_degrees**

Read only. Same as *absolute_rotation* but returns degrees.

Node.**scale**

Gets or sets the node scale, as *geometry.Vector*.

IMPORTANT: Node scale is always get or set relative to its parent node. To get the absolute scale, use the *absolute_scale* property.

The x value of the vector represents scaling in the X axis, while y value is for scaling in the Y axis. Negative values of x or y are possible - it will make the node to be rendered as a mirror reflection in X and/or Y axis respectively.

```
import math
from kaa.nodes import Node
from kaa.geometry import Vector

# inside a Scene's __init__ :
self.node1 = Node(position = Vector(100, 100))
self.root.add_child(self.node1)
self.node1.scale = Vector(2, 0.5)  # stretch the node by a factor of 2 in the X␣
↪axis and shrink it by a factor of 0.5 in the Y axis

# add a child node
self.node2 = Node(position=Vector(-5, -15), scale=Vector(4, 0.5))
self.node1.add_child(self.node2)

print(self.node1.scale)  # V[2.0, 0.5]
print(self.node2.scale)  # V[4.0, 0.5]
print(self.node2.absolute_scale)  # V[8.0, 0.25]
```

Node.**absolute_scale**
Read only. Returns an absolute scale, as *geometry.Vector*. Check out the example in the *scale* property section.

Node.**visible**
Gets or sets the visibility of the node (shows or hides it), using bool.

Makes most sense for nodes which are rendered on the screen such as nodes having sprites, or text nodes.

Note that this has only a visual effect, so for example setting `visible` to `False` on a *physics.HitboxNode* will not make the hitbox inactive - it will still detect collisions normally.

Setting visible to `False` will hide all of its child nodes (recursively) as well.

Node.**sprite**
Gets or sets a *sprites.Sprite* for the node.

A sprite is an immutable object that wraps a graphical image.

Assigning a Sprite to a Node will make the sprite be displayed at node's position, with node's rotation and scale.

Creating a frame by frame animation is a two step process:

First you'll need to have a list of frames, each frame being an individual *sprites.Sprite* instance. You can load each frame from a separate file or, if you have a spritesheet (a single graphical file which includes all frames) use the utility function *sprites.split_spritesheet()* to cut the sprites out of the file.

Second, you'll need to create a *transitions.NodeSpriteTransition* transition using the list of sprites, which also allows you to specify the animation duration, looping etc. and *assign that transition to the node*

Note: Transitions are a more general mechanism than just sprite animations. *Read more about transitions here.*.

Since sprite is a dimensional object (has its width and height) and node position is just a 2D (x, y) coords, it is important to understand the concept of node's origin point. Read more about *Node origin points*.

Example 1 - a node with a static sprite.

```
from kaa.nodes import Node
from kaa.sprites import Sprite
```

(continues on next page)

(continued from previous page)

```
from kaa.geometry import Vector, Alignment
import os

# inside a Scene's __init__ :
my_sprite = Sprite(os.path.join('assets', 'gfx', 'arrow.png'))  # create a sprite␣
↪from image file
self.node = Node(position=Vector(100, 100), sprite=my_sprite))  # create a Node␣
↪at (100, 100) with the sprite
self.node.origin_alignment = Alignment.center # this makes the (100, 100)␣
↪position be at the center of the sprite
self.root.add_child(self.node)  # until you add the Node to the Scene it won't␣
↪show up on the screen!
```

Example 2 - a node with frame by frame animation running in an infinite loop:

```
from kaa.nodes import Node
from kaa.sprites import Sprite
from kaa.geometry import Vector
from kaa.transitions import NodeSpriteTransition

# inside a Scene's __init__:
spritesheet = Sprite(os.path.join('assets', 'gfx', 'spritesheet.png'))  # a␣
↪1000x1000 spritesheet with hundred 100x100 frames
frames = split_spritesheet(spritesheet, Vector(100,100)) # cut the spritesheet␣
↪into 100 individual <Sprite> instances
animation = NodeSpriteTransition(frames, duration=2., loops=0, back_and_
↪forth=False) # With 100 frames a duration of 2 secs means 20 miliseconds per␣
↪frame.
self.node = Node(position=Vector(100, 100), transition=animation)  # the␣
↪transition will take care of setting the appropriate <Sprite> over time, thus␣
↪creating an animation effect.
self.root.add_child(self.node)  # until you add the Node to the Scene it won't␣
↪show up on the screen!
```

To stop playing an animation simply set the node's transition to `None`

`Node.`**`color`**

Gets or sets the color of the shape of the node, using *`colors.Color`*.

In practice, if a node has a sprite that means that a sprite will be tinted in that color.

If a node does not have a sprite it still can have a shape (see the *shape* property). In that case setting a color will make the shape be rendered in that color.

For text nodes (*`fonts.TextNode`*) it gets or sets the color of the text.

It is often useful to set a color for hitbox nodes (*`physics.HitboxNode`*) to see where the hitboxes are in relation to the node's sprite. Just remember to set a high enough z_index on the hitbox node.

The default color of a Node is a "transparent" color (r=0, g=0, b=0, a=0).

`Node.`**`shape`**

Gets or sets a shape of a Node. A shape can be one of the following types:

- `None` - this is the default value (no shape)
- *`geometry.Circle`* - the shape has a form of a circle
- *`geometry.Polygon`* - the shape has a form of a polygon.

The most common scenario for setting a shape manually is for the hitbox nodes (*physics.HitboxNode*). It defines an area that will generate collisions. More information is available in the *physics module documentation*).

If you set a Sprite for a Node, its shape will be automatically set to a rectangular polygon corresponding with the size of the sprite.

Overriding sprite node's shape is usually not necessary, but you can always do that. For example, you can set a 100x200 px sprite for a node and then set a custom shape e.g. a non-rectangular polygon or a circle. The drawn image will be fit inside the defined shape.

Node.**origin_alignment**
Gets or sets origin alignment of a node, as *geometry.Alignment*.

It's best to show what origin point is on an example. Assume you have a Node with a 100x50 px *sprite*. You tell the engine to draw the node at some specific position e.g. `position=Vector(300, 200)`. But what does this actually mean? Which pixel of the 100x50 image will really be drawn at (300, 200)? The top-left pixel? Or the central pixel? Or maybe some other pixel?

By default it's the central pixel and that reference point is called the 'origin'. By setting the origin_alignment you can change the position of the point to one of the 9 default positions: from top left, through center to the bottom right.

Setting the origin alignment is especially useful when working with text nodes (*fonts.TextNode*) as it allows you to align text to the left or right.

If you need a custom origin point position, not just one of the 9 default values, you can always wrap a node with a parent node. Remember that node positions are always set in relation to their parents, so by creating a parent-child node relations and setting origin_alignment appropriately, you can lay out the nodes on the scene any way you want.

Node.**lifetime**
Gets or sets a lifetime of the node, in seconds.

By default nodes live forever. After you add them to the scene with *Node.add_child()* method they will stay there until you delete them by calling *Node.delete()*.

Setting the lifetime of a node will remove the node automatically from the scene after given number of seconds. It's important to note that the timer starts ticking after you add the node to the scene, not when you instantiate the node.

Node.**transition**
Gets or sets a default transition object.

Transitions are "recipes" how the node's properties (such as position, rotation, scale, color, sprite, etc.) should evolve over time. Transitions system is a very powerful feature, *refer to transitions documentation for details*.

Node.**transitions_manager**
Read only. Returns a *transitions.NodeTransitionsManager* object which allows you to manage multiple transitions on a Node.

Transitions are "recipes" how the node's properties (such as position, rotation, scale, color, sprite, etc.) should evolve over time. Transitions system is a very powerful feature, *refer to transitions documentation for details*.

Node.**absolute_transformation**
Gets the absolute transformation of the Node, in form of a *geometry.Transformation* instance.

Node.**transformation**
Gets or sets the transformation of the Node, in form of a *geometry.Transformation* instance. Applying a transformation to the node is an equivalent of changing its *position* (translate Transformation), *rotation* (rotating Transformation) or *scale* (scaling Transtofmation). Refer to *geometry.Transformation* for more details on how to work with transformation objects.

Node.**views**
Gets or sets indexes of views (as a set object) in which this node shall be rendered. Each scene can have a maximum of 32 views (indexed -16 to 15). Default value is None meaning the node will inherit the view from its parent, and to find the actual `views` value use *effective_views* property.

Note that the *root node of the scene* has a view set to {0} (a set with just one element: zero) by default, so all nodes added to root (and their children) will have a views value set to {0}. Read more about views in *engine.View* reference.

```
self.root.add_child(Node(position=Vector(32,45), sprite=some_sprite))  # will be␣
↪rendered in the default view
self.root.add_child(Node(position=Vector(-432,-445), sprite=some_sprite, views={0,
↪ 1, 15}))  # will be rendered in views 0, 1 and 15
```

```
node = Node(views={13})  # node will be rendered in view 13
child = Node()
node.add_child(child)  # the child will also be rendered in view 13
```

Node.**effective_views**
Gets effective views value of the node. Use it to find the actual views value when node inherits it from its parent.

```
# ... inside a Scene class...

node = Node(views={1, 3})
child = Node()  # views is None

node.add_child(child)
self.root.add_child(node)

print(child.views)  # prints None
print(child.effective_views)  # prints {1, 3}
```

Node.**indexable**
Gets or sets whether the node is indexable (as `bool`). Default is `True`. If set to *True*, this Node will be queryable by the *engine.SpatialIndexManager*.

Setting this value to False yields a slight performance boost.

Node.**bounding_box**
Returns node's bounding box as *geometry.BoundingBox*. Bounding box is X/Y axis - aligned rectangle that contains *Node's shape*.

Instance Methods:

Node.**add_child**(*child_node*)
Adds a child node to the current node. The child_node must be a *Node* type or subtype.

Each Scene always has a *root node*, which allows to add your first nodes.

When a parent node gets transformed (repositioned, scaled, rotated), all its child nodes are transformed accordingly.

You can build the node tree freely, with some exceptions:

- *physics.BodyNode* must be a direct child of a *physics.SpaceNode*
- *physics.HitboxNode* must be a direct child of a *physics.BodyNode*

Node.**delete**()
Deletes a node from the scene. All child nodes get deleted automatically as well.

Important: The node gets deleted immediately so you should not read any of the deleted node's properties afterwards. It may result in segmentation fault error and the whole process crashing down.

See also: *Node lifetime*

`Node.`**`get_relative_position`**`(`*ancestor*`)`
Returns node's position (*`geometry.Vector`*) relative to given ancestor.

The `ancestor` parameter must be a *`Node`* and it must be an ancestor of a node on which the method is called.

`Node.`**`get_relative_transformation`**`(`*ancestor*`)`
Returns node's transformation (`geomtry.Transformation`) relative to given ancestor.

The `ancestor` parameter must be a *`Node`* and it must be an ancestor of a node on which the method is called.

`Node.`**`on_detach`**`()`
You don't call this method directly. Instead you can implement it on a class that inherits from Node. The method gets called when the node is removed from the scene (by calling delete(), its lifetime expiring, scene being destroyed, and so on...).

Once the node gets removed, accessing its properties results in an error, so `on_detach()` offers an opportunity to execute some cleanup code.

```
class MyBulletNode(Node):

    def on_detach(self):
        # remove the node from our own custom collection
        self.scene.my_bullets_manager.remove_bullet(self)
```

`Node.`**`on_attach`**`()`
You don't call this method directly. Instead you can implement it on a class that inherits from Node. The method gets called when the node is added to the scene.

```
class MyBulletNode(Node):

    def on_attach(self):
        # add the node to our own custom collection
        self.scene.my_bullets_manager.add_bullet(self)
```

`Node.`**`__bool__`**`(`*self*`)`
Allows to inspect the node to verify if it's in a valid state.

```
# ... inside a scene ....
node = Node()
self.root.add_child(node)
assert node
node.delete()
assert not node
```

2.10 `physics` — A 2D physics system, with rigid bodies, collisions and more!

Kaa inlcudes a 2D physics engine which allows you to easily add physical features to objects in your game, handle collisions etc. The idea is based on three types of specialized *nodes*:

- *`SpaceNode`* - it represents physical simulation environment, introducing environmental properties such as gravity or damping.

- *BodyNode* - represents a physical body. Each BodyNode must be a direct child of a *physics.SpaceNode*. BodyNode can have HitboxNodes as child nodes.
- *HitboxNode* - represents an area of a BodyNode which can collide with other HitboxNodes. Must be a direct child of a *physics.BodyNode*.

Read more about *the nodes concept in general*.

Note: Physics system present in the kaa engine is a wrapper of an excellent 2D physics library - Chipmunk.

2.10.1 SpaceNode reference

class physics.**SpaceNode**(*gravity=Vector(0, 0)*, *damping=1*, *position=Vector(0, 0)*, *rotation=0*, *scale=Vector(1, 1)*, *z_index=0*, *color=Color(0, 0, 0, 0)*, *sprite=None*, *shape=None*, *origin_alignment=Alignment.center*, *lifetime=None*, *transition=None*, *visible=True*)

SpaceNode extends the *nodes.Node*. It represents physical simulation environment. All BodyNodes must be direct children of a SpaceNode. Typically you'll need just one SpaceNode per Scene, but nothing prevents you from adding more. If you decide to use multiple SpaceNodes on a Scene, be aware that they will be isolated (BodyNodes under SpaceNode A won't be able to interact with BodyNodes under SpaceNode B).

Space node is also place where you can register collision handlers (see *SpaceNode.set_collision_handler()*). Collision handlers are your custom functions which will be called when a collision between a pair of defined hitbox nodes occurs.

Another feature of the SpaceNode is running spatial queries. You can find hitboxes colliding with a custom shape (*geometry.Circle*, *geometry.Polygon* or *geometry.Segment*) via *SpaceNode.query_shape_overlaps()*. You can find hitboxes colliding with a ray cast between points A and B using *SpaceNode.query_ray()*. Finally you can also find hitboxes around a specific point with `SpacenNode.query_point_neighbors()`.

Constructor accepts all parameters from the base *nodes.Node* class and adds the following new parameters:

- `gravity` - a *geometry.Vector*
- `damping` - a number

Instance properties:

`SpaceNode.`**`gravity`**

Gets or sets the gravity inside the SpaceNode, as *geometry.Vector*. Direction of the vector determines the direction of the gravitational force, while it's length determines gravity strength.

Gravity will be applied only to the dynamic BodyNodes. Kinematic and Static BodyNodes do not have mass and therefore are not affected by the gravity.

Default gravity is zero, meaning no gravitational forces applied.

`SpaceNode.`**`damping`**

Gets or sets the damping inside the SpaceNode. Represents a "friction" or a "drag force" inside the environment which slows all BodyNodes down with time. A damping of 0.25 means velocity of all BodyNodes will decrease by a factor of 4 in 1 second. A damping of 1 (default) means no slowdown force applied. A damping greater than 1 will make all BodyNodes accelerate, proportionally to its value.

Damping is applied only to the dynamic BodyNodes. Kinematic and Static BodyNodes do not have mass and therefore ignore the damping effect.

`SpaceNode.`**`sleeping_threshold`**

Gets of sets the sleep time threshold (in seconds) which affects all BodyNodes in the SpaceNode. If given

BodyNode remains static (doesn't change its position or rotation) for that amount of time the engine will stop making physical calculations for it. In some situations it can improve the performance. A body remaining in a sleeping state can still collide with other bodies - that will force it to move and 'wake up' as a consequence.

Default value for the sleeping_threshold is infinite, which effectively means that the performance mechanism is disabled.

Instance methods:

`SpaceNode.`**`set_collision_handler`**(*trigger_a*, *trigger_b*, *handler_callable*)

Registers a custom collision handler function between two *`HitboxNode`* instances, tagged with trigger_a and trigger_b respectively. The function will get called when collision between hitboxes occur.

Note, that collisions occur between HitboxNodes (not between BodyNodes!). The `trigger_a` and `trigger_b` params are your own values which you use to tag *`HitboxNode`*. They should be of integer type.

`handler_callable` is your own callable, it takes the following three parameters:

- `arbiter` - an *`Arbiter`* object that holds additional information about collision.
- `collision_pair_a`- a *`CollisionPair`* object that allows identifying which BodyNode and which HitboxNoded collided. Corresponds with HitboxNode identified by trigger_a.
- `collision_pair_b`- a *`CollisionPair`* object that allows identifying which BodyNode and which HitboxNoded collided. Corresponds with HitboxNode identified by trigger_b.

If your collision handler function does not return any value, the collision will occur normally. However if you return 0 in the collision handler AND you do that in the begin or pre_solve phase, then collision will be ignored by the physics engine (no impulses will be applied to colliding objects).

```
# somwhere in the code...
bullet_hitbox = HitboxNode(shape=Circle(radius=10), trigger_id=123, ..... )  #␣
↪123 is our own value we give to all bullet hitboxes
enemy_hitbox = HitboxNode(shape=Circle(radius=10), trigger_id=456, ..... )  #␣
↪456 is our own value we give to all enemy hitboxes

# collision handler function:
def on_collision_bullet_enemy(arbiter, bullet_pair, enemy_pair):
    print("Detected a collision between a bullet object's {} hitbox {} and Enemy
↪'s object {} hitbox {}".format(
        bullet_pair.body, bullet_pair.hitbox, enemy_pair.body, enemy_pair.hitbox))
    # ... write code to handle the collision effects ....

# assuming space_node is <SpaceNode>,
# 123 and 456 here are defining which pair of hitbox collisions shall be handled␣
↪by the on_collision_bullet_enemy
# in this case it defines a pair of a bullet hitbox and enemy hitbox
space_node.set_collision_handler(123, 456, on_collision_bullet_enemy)
```

IMPORTANT: Collision handler function can be called multiple times for given pair of colliding objects (even multiple times per frame). This can happen if object's hitboxes touch for the first time, then they either overlap or touch each other for some time and finally - they separate. The collision handler function will be called every frame, as long as the hitboxes touch or overlap. When they make apart, the collision handler function stops being called.

`SpaceNode.`**`query_shape_overlaps`**(*shape*, *mask=kaa.physics.COLLISION_BITMASK_ALL*, *collision_mask=kaa.physics.COLLISION_BITMASK_ALL*, *group=kaa.physics.COLLISION_GROUP_NONE*)

Takes a shape (*`geometry.Circle`* or *`geometry.Polygon`*) and returns hitboxes which overlap with that

shape (either partially or entirely) as well as body nodes which own those hitboxes. The shape coordinates are expected to be in a frame reference relative to the SpaceNode.

When running the query, the shape you pass is treated like a hitbox node, therefore parameters such as mask, collision_mask and group behave identically as in *HitboxNode*. It means you can use those params for filtering purpose. Refer to *mask*, *collision_mask* and *group* for more information.

The query returns a list of *ShapeQueryResult* objects. Each *ShapeQueryResult* represents a 'collision' of the shape with one hitbox. It holds a reference to hitbox' parent (body node) and other metadata such as intersection points.

```
from kaa.physics import SpaceNode, BodyNode, HitboxNode
from kaa.geometry import Polygon

self.space = SpaceNode()
self.root.add_child(self.space)
body_node = BodyNode(position=Vector(0, 0))
hitbox = HitboxNode(shape=Polygon.from_box(Vector(100, 100)))
body_node.add_child(hitbox)
self.space.add_child(body_node)
# find hitboxes intersecting with our triangular polygon
triangle = Polygon([Vector(0, 0), Vector(100, 100), Vector(0, 200) ])
results = self.space.query_shape_overlaps(triangle)
for result in results:
    print(f"Shape {triangle.points} collided with hitbox {result.hitbox.shape.
↪points} owned "
          f"by {result.body}. Contact points metadata accessible at {result.
↪contact_points}.")
```

SpaceNode.**query_ray**(*ray_start*, *ray_end*, *radius=0.*, *mask=kaa.physics.COLLISION_BITMASK_ALL*, *collision_mask=kaa.physics.COLLISION_BITMASK_ALL*, *group=kaa.physics.COLLISION_GROUP_NONE*)

A "ray casting" method. Takes in a ray (two Vectors: `ray_start` and `ray_end`) and returns hitboxes (and their owning BodyNodes) which collide with that ray. The ray coordinates are expected to be in a frame reference relative to the SpaceNode.

The `radius` parameter sets the width of the cast ray.

When running the query, the ray is treated like a hitbox node, therefore parameters such as mask, collision_mask and group behave identically as in *HitboxNode*. It means you can use those params for filtering purpose. Refer to *mask*, *collision_mask* and *group* for more information.

The query returns a list of *RayQueryResult* objects. Each represents a collision of the ray with one hitbox. It holds a reference to hitbox owner (body node) and other metadata such as intersection point.

```
from kaa.physics import SpaceNode, BodyNode, HitboxNode
from kaa.geometry import Polygon

self.space = SpaceNode()
self.root.add_child(self.space)
body_node = BodyNode(position=Vector(0, 0))
hitbox = HitboxNode(shape=Polygon.from_box(Vector(100, 100)))
body_node.add_child(hitbox)
self.space.add_child(body_node)

# cast a ray and find hitboxes colliding with the ray
results = self.space.query_ray(ray_start=Vector(-200, -200), ray_end=Vector(200,
↪200))
for result in results:
```

(continues on next page)

(continued from previous page)

```
    print(f"Ray collided with {result.hitbox.shape.points} hitbox owned by
↪{result.body} at "
          f"{result.point}. Normal was {result.normal} and alpha was {result.
↪alpha}")
```

SpaceNode.**query_point_neighbors**(*point*, *max_distance*, *mask=kaa.physics.COLLISION_BITMASK_ALL*, *collision_mask=kaa.physics.COLLISION_BITMASK_ALL*, *group=kaa.physics.COLLISION_GROUP_NONE*)

Queries for hitboxes `max_distance` away from `point`. The `point` must be a *geometry.Vector*.

When running the query, the `point` is treated like a hitbox node, therefore parameters such as mask, collision_mask and group behave identically as in *HitboxNode*. It means you can use those params for filtering purpose. Refer to *mask*, *collision_mask* and *group* for more information.

The query returns a list of *PointQueryResult* objects which contain collision data such as references to hitbox, its owner body node and other metadata.

```
from kaa.physics import SpaceNode, BodyNode, HitboxNode
from kaa.geometry import Polygon

self.space = SpaceNode()
self.root.add_child(self.space)
body_node = BodyNode(position=Vector(0, 0))
hitbox = HitboxNode(shape=Polygon.from_box(Vector(100, 100)))
body_node.add_child(hitbox)
self.space.add_child(body_node)

# find hitboxes in the vicinity of a point
point = Vector(-140, 140)
results = self.space.query_point_neighbors(point=point, max_distance=200)
for result in results:
    print(f"Point {point} collided with hitbox {result.hitbox.shape.points} owned
↪"
          f"by {result.body}. Collision point is at {result.point}, distance:
↪{result.distance}")
```

2.10.2 BodyNode reference

class physics.**BodyNode**(*body_type=BodyNodeType.dynamic*, *force=Vector(0, 0)*, *velocity=Vector(0, 0)*, *mass=20.0*, *moment=10000.0*, *torque=0*, *torque_degrees=0*, *angular_velocity=0*, *angular_velocity_degrees=0*, *position=Vector(0, 0)*, *rotation=0*, *scale=Vector(1, 1)*, *z_index=0*, *color=Color(0, 0, 0, 0)*, *sprite=None*, *shape=None*, *origin_alignment=Alignment.center*, *lifetime=None*, *transition=None*, *visible=True*)

BodyNode extends the *nodes.Node* class, introducing physical features.

In the nodes tree, BodyNode must be a direct child of a *SpaceNode*.

BodyNode is the only node type which can have *HitboxNode* as children nodes.

BodyNodes themselves never collide with each other. The need to have HitboxNodes as children to generate collisions. A BodyNode can have multiple HitboxNodes.

BodyNode constructor accepts all parameters from the base *nodes.Node* class and adds the following new parameters:

- `body_type` - a *BodyNodeType* enum value. *Learn more here*

- `force` - a *geometry.Vector*
- `velocity` - a *geometry.Vector*
- `mass` - a number
- `moment` - a number
- `torque` - a number
- `torque_degrees` - a number, alternative to `torque`, using degrees instead of radians
- `angular_velocity` - a number
- `angular_velocity_degrees` - a number, alternative to `angular_velocity`, using degrees instead of radians

Instance properties:

`BodyNode`**`.body_type`**

Gets or sets body type, must be a *BodyNodeType* value. There are three types available:

- static - the body has infinite mass and won't move when its hitboxes collide with any other hitboxes. You cannot move it "manually" by setting its velocity or angular velocity either. Those nodes are **truly** static.
- kinematic - similar to static body in a sense that its velocity or rotation will never be affected by anything, e.g. its hitboxes colliding. But the difference is that you can move and rotate that type of body. The collisions will occur normally and you will be able to handle them.
- dynamic - the default type. Physics engine will calculate body's velocity and angular velocity when its hitboxes will collide with other bodies' hitboxes.

Use static bodies for static obstacles and other elements on the scene that you know won't move, but you want them to collide with other bodies and block their movement. Those bodies will always have zero velocity and zero angular velocity.

Use kinematic bodies for objects which you want to move but you don't want their velocity controlled by the physics engine. Those nodes won't move or rotate on their own. The onus is on you to set their velocity or angular velocity but you still want to be able to detect collisions between them and other objects on the scene.

Use dynamic bodies for freely moving objects that you want physics engine to fully take care of. Dynamic bodies have their velocity and angular velocity calculated by the engine.

Note: Example: a classic space shooter Git Gud or Get Rekt, built with kaa engine is using kinematic bodies for player, enemies, and bullets, and dynamic bodies for debris left on the scene after enemies explode.

`BodyNode`**`.force`**

Gets or sets a custom force applied to the BodyNode, as *geometry.Vector*. The force is reset to zero on each frame, so if you want it to constantly work on the object, you need to apply it on each frame.

Applying force affects object's velocity.

Force has an effect only on *dynamic body nodes*. Static and kinematic body nodes will not be affected.

`BodyNode`**`.local_force`**

Same as *BodyNode.force* but uses strictly local frame of reference.

```
node.rotation_degrees = 0
node.force = Vector(1, 0)  # force will drag the object in direction V(1, 0),␣
↪regardless to node rotation
```

(continues on next page)

(continued from previous page)

```
other_node.rotation_degrees = 45
other_node.local_force = Vector(1, 0)  # force direction will be calculated AFTER␣
↪applying the rotation!
```

BodyNode.**velocity**
Gets or sets the linear velocity of the BodyNode, as *geometry.Vector*. Linear velocity vector determines the speed and direction of movement of an object.

For *dynamic body nodes* the velocity is calculated by the physics engine. You can override the velocity value calculated by the engine but you should consider *applying force* instead.

Setting velocity from your code is recommended for kinematic bodies, as they won't move on their own otherwise.

BodyNode.**mass**
Gets or sets the mass for the body node. Mass has an effect on the output velocity of dynamic body when it collides with other bodies.

BodyNode.**torque**
Gets or sets the torque for the body node. Using radians. The torque is reset to zero on each frame, so if you want it to constantly work on the object you need to apply it on each frame.

Applying torque affects object's angular velocity.

Applying torque has an effect only on *dynamic body nodes*. Static and kinematic body nodes are not affected.

For degrees use *torque_degrees*

BodyNode.**torque_degrees**
Gets or sets the torque for the body node. Using degrees. See *torque*

BodyNode.**angular_velocity**
Gets or sets the angular velocity for the body node. Using radians. Angular velocity determines how fast the object rotates and the direction of the rotation (clockwise or anticlockwise).

Similarly to *velocity* the angular velocity is calculated by the physics engine for *dynamic body nodes*. You can override the angular velocity manually but you should consider *applying torque* instead.

Setting angular velocity from your code is recommended for kinematic bodies, as they won't rotate on their own otherwise.

For degrees use *angular_velocity_degrees*

BodyNode.**angular_velocity_degrees**
Gets or sets the angular velocity for the body node. Using degrees. See *angular_velocity*

BodyNode.**moment**
Gets or sets the moment for the body node. Moment has an effect on the output angular velocity of dynamic body when it collides with other bodies.

BodyNode.**sleeping**
Gets or sets the sleeping status of the node as bool. If set to `True` it gives the physics engine a performance hint, making it ignore this node when calculating its velocity and angular velocity. The node will wake up automatically when it's moving or rotating so it doesn't makes sense to set the sleeping status on a moving or rotating nodes.

See also: *SpaceNode.sleeping_threshold*.

Instance methods:

BodyNode.**apply_force_at_local**(*force*, *at*)
Applies `force` (*geometry.Vector*) to this body node at position `at` (*geometry.Vector*). The `at`

parameter is in a relative frame of reference. For example, if `at` is `Vector(0, 0)` then the force will be applied at the center of the body node.

Note: Applied force will be automatically reset to zero each frame, so if you want to apply force constantly you should do that on each frame.

`BodyNode.`**`apply_impulse_at_local`**(*impulse*, *at*)

Applies `impulse` (*`geometry.Vector`*) to this body node at position `at` (*`geometry.Vector`*). The `at` parameter is in a relative frame of reference. For example, if `at` is `Vector(0, 0)` then the impulse will be applied at the center of the body node.

Note: Use impulses when you need to apply a very large force applied over a very short period of time. Some examples are a ball hitting a wall or cannon firing.

`BodyNode.`**`apply_force_at`**(*force*, *at*)

Same as *`BodyNode.apply_force_at_local()`* but `at` is in an absolute frame of reference. For instance, if body node's *absolute position* is Vector(110, 34) and you want to apply the force at the center of the body, you need to pass `at=Vector(110, 34)`.

`BodyNode.`**`apply_impulse_at`**(*impulse*, *at*)

Same as *`BodyNode.apply_impulse_at_local()`* but `at` is in an absolute frame of reference. For instance, if body node's *absolute position* is Vector(110, 34) and you want to apply the impulse at the center of the body, you need to pass `at=Vector(110, 34)`.

2.10.3 `HitboxNode` reference

`class` `physics.`**`HitboxNode`**(*shape*, *group=kaa.physics.COLLISION_GROUP_NONE*, *mask=kaa.physics.COLLISION_BITMASK_ALL*, *collision_mask=kaa.physics.COLLISION_BITMASK_ALL*, *trigger_id=None*, *position=Vector(0, 0)*, *rotation=0*, *scale=Vector(1, 1)*, *z_index=0*, *color=Color(0, 0, 0, 0)*, *sprite=None*, *shape=None*, *sensor=False*, *elasticity=0.95*, *friction=0*, *surface_velocity=Vector(0, 0)*, *origin_alignment=Alignment.center*, *lifetime=None*, *transition=None*, *visible=True*))

HitboxNode extends the `Node` class and introduces collision detection features.

In the nodes tree, HitboxNode must be a direct child of a *`BodyNode`*. A *`BodyNode`* can have many HitboxNodes.

HitboxNode inherits all `Node` properties and methods, some of which may be particularly useful for debugging. For example, by setting a color and z_index of on a HitboxNode you can make the hitbox visible.

Hitbox node has its own specific params, related with collision handling:

- `shape` - can be either *`geometry.Polygon`* or *`geometry.Circle`*
- `group` - an integer, default value is a kaa constant meaning "no group". Hitboxes within the same group will never collide with each other.
- `mask` - an integer, used as a bit mask, it's recommended to use enum.Intflag enumerated constant. Default value is a kaa constant meaning "match all masks". Defines a category of this hitbox.
- `collision_mask` - an integer, used as a bit mask, it's recommended to use enum.Intflag enumerated constant. Default value is a kaa constant meaning "match all masks". Defines with which categories this hitbox should collide.

- `trigger_id` - an integer, your own value used with the *SpaceNode.set_collision_handler()* method. Used in custom collision handling.

The hitbox node also has a few properties affecting its physical behaviour:

- `sensor`
- `elasticity`
- `friction`
- `surface_velocity`

Instance properties:

`HitboxNode.`**`shape`**
Gets or sets the shape of the hitbox. It can be either *geometry.Polygon* or *geometry.Circle*.

`HitboxNode.`**`group`**
Gets or sets the group of the hitbox, as integer. Hitboxes with the same group won't collide with each other. It's basically a performance hint for the physics engine. Default value is kaa.physics.COLLISION_GROUP_NONE, meaning no group is used.

Another method of telling the engine which hitbox collisions it should ignore is to set `mask` and `collision_mask` on a HitboxNode.

`HitboxNode.`**`mask`**
Gets or sets the category of this hitbox node, as a bit mask. Other nodes will collide with this node if they match on collision_mask. Otherwise collisions will be ignored. Use mask and collision_mask as performance hints for the engine.

By default mask and hitbox_mask are kaa.physics.COLLISION_BITMASK_ALL which meaning the engine will not apply any filtering when detecting collisions - hitbox with those values will collide with any other hitbox.

An example below shows how to set mask and collision_mask values to apply the following logic:

- player hitbox will collide only with enemy hitbox, enemy bullet hitbox and wall hitbox
- player bullet hitbox will collide only with the enemy hitbox
- enemy hitbox will collide only with other enemy hitboxes, player, player bullet and wall hitbox
- enemy bullet will collide only with the player hitboxes
- wall will collide with everything except other wall hitboxes

```
from kaa.physics import HitboxNode
from kaa.geometry import Circle, Vector, Polygon
import enum

class CollisionMask(enum.IntFlag):
    player = enum.auto()
    player_bullet = enum.auto()
    enemy = enum.auto()
    enemy_bullet = enum.auto()
    wall = enum.auto()

    player_collision_mask = enemy | enemy_bullet | wall
    enemy_collision_mask = enemy | player | player_bullet | wall
    wall_collision_mask = player | player_bullet | enemy | enemy_bullet

player_hitbox = HitboxNode(shape=Circle(radius=20), mask=CollisionMask.player,
```

(continues on next page)

(continued from previous page)

```
                              collision_mask=CollisionMask.player_collision_mask)
player_bullet_hitbox = HitboxNode(shape=Circle(radius=5), mask=CollisionMask.
↪player_bullet,
                                  collision_mask=CollisionMask.enemy)
enemy_hitbox = HitboxNode(shape=Circle(radius=20), mask=CollisionMask.enemy,
                          collision_mask=CollisionMask.enemy_collision_mask)
enemy_bullet_hitbox = HitboxNode(shape=Circle(radius=5), mask=CollisionMask.enemy_
↪bullet,
                                 collision_mask=CollisionMask.player)
wall = HitboxNode(shape=Polygon([Vector(-50, -50), Vector(-50, 50), Vector(0,␣
↪100)],
                  mask=CollisionMask.wall, collision_mask=CollisionMask.wall_
↪collision_mask))
```

What if there's assymetry in the mask and collision_mask definitions? For example, what will happens if we set the player to collide with enemy, but won't set enemy to collide with the player? In that case, those collisions won't occur. The collision masks need to match symmetrically from both sides for collision to be detected.

What if there is a proper symmetry in collision mask definitions but both hitboxes have the same *group*? In that case the group value takes precedence and collisions won't occur.

`HitboxNode.`**`collision_mask`**
Gets or sets the categories of other hitboxes that you want this hitbox to collide with.

See the full example in the *mask* section above for more information.

`HitboxNode.`**`trigger_id`**
Gets or sets the trigger id value. It can be any value of your choice. It's a 'tag' value which you need to pass when *registering your custom collision handler function*

`HitboxNode.`**`sensor`**
Gets or sets the sensor flag (bool). Default is `False`. If set to `True`, the hitbox will not cause any physical collision effects (i.e. will not interact with other colliding objects) but will still trigger its collision handler function (check out *SpaceNode.set_collision_handler* method for more info on how to register a collision handlers for hitboxes).

`HitboxNode.`**`elasticity`**
Gets or sets hitbox elasticity, as `float`. This is a percentage of kinetic energy transferred during collision and should be between 0 and 1. A value of 0.0 gives no bounce, while a value of 1.0 will give a "perfect" bounce. Default elasticity is 0.95. The elasticity for a collision is found by multiplying the elasticity of the interacting hitboxes together.

`HitboxNode.`**`friction`**
Gets or sets hitbox friction coefficient, as `float`. Physics engine uses the Coulomb friction model, a value of 0.0 is frictionless. The friction for a collision is found by multiplying the friction of the interacting hitboxes together. Default is 0.

`HitboxNode.`**`surface_velocity`**
Gets or sets hitbox surface velocity, as *`geometry.Vector`*. Useful for creating conveyor belts or players that move around. This value is only used when calculating friction, not resolving the collision. Default is `Vector(0, 0)` (no surface velocity)

2.10.4 `ShapeQueryResult` reference

`class` `physics.`**`ShapeQueryResult`**
ShapeQueryResult object is returned by the *`SpaceNode.query_shape_overlaps()`* method. A single query can return multiple ShapeQueryResult objects. A ShapeQueryResult has the following properties:

- `hitbox` - an instance of *`HitboxNode`* which collided
- `body` - a *`BodyNode`* instance that owns the hitbox
- `contact_points` - a list of *`CollisionContactPoint`* objects which contain information about collision points

2.10.5 `CollisionContactPoint` reference

class physics.**CollisionContactPoint**

A CollisionContactPoint instance represents an actual point where collision between two shapes occurred. It has the following properties:

- `point_a`
- `point_b`
- `distance`

2.10.6 `Arbiter` reference

class physics.**Arbiter**

Arbiter object is passed to the collision handler function when collision occurs. It holds information about the collision in following fields:

- `space` - a *`SpaceNode`* where collision occurred.
- `phase` - an enum value (*`CollisionPhase`*), indicating collision phase. Available values are:
  - `CollisionPhase.begin` - indicates that collision betwen two objects has started (their hitboxes have just touched or overlapped)
  - `CollisionPhase.pre_solve` - indicates that two hitboxes are still in contact (touching or overlapping). It is called before the engine calculates the physics (e.g. velocities of both colliding objects)
  - `CollisionPhase.post_solve` - like pre_solve, but called after the engine calculates the physics for the objects.
  - `CollisionPhase.separate` - indicates that hitboxes of our two objects have separated - the collision has ended

2.10.7 `CollisionPair` reference

class physics.**CollisionPair**

CollisionPair object is passed to the collision handler function (see *`SpaceNode.set_collision_handler()`*). It holds references to an object that collided. The CollisionPair has the following fields:

- `body` - referencing *`BodyNode`* which collided
- `hitbox` - referencing *`HitboxNode`* which collided. Note that body nodes can have multiple hitboxes: here you can find which of them has collided

2.10.8 `BodyNodeType` reference

class physics.**BodyNodeType**

Enum type used for classifying BodyNodes. It has the following values:

- `BodyNodeType.static`
- `BodyNodeType.dynamic`
- `BodyNodeType.kinematic`

Refer to BodyNode's *body_type property* for more information.

2.10.9 `CollisionPhase` reference

class physics.**CollisionPhase**

Enum type used by the collision handler *Arbiter*. It has the following values:

- `CollisionPhase.begin`
- `CollisionPhase.pre_solve`
- `CollisionPhase.post_solve`
- `CollisionPhase.separate`

2.10.10 `RayQueryResult` reference

class physics.**RayQueryResult**

RayQueryResult objects are returned by the *SpaceNode.query_ray()* method. A ShapeQueryResult represents a collision between a ray and a hitbox. It has the following properties:

- `hitbox` - an instance of *HitboxNode* which collided
- `body` - a *BodyNode* instance that owns the hitbox
- `point` - a *geometry.Vector* where the ray intersected the hitbox
- `normal` - a *geometry.Vector* with ray reflection direction. This vector is normalized.
- `alpha` - a float number indicating distance from the ray start point to the point where collision occurred. The distance is in relation to the ray length so the number is always between 0 and 1.

2.10.11 `PointQueryResult` reference

class physics.**PointQueryResult**

PointQueryResult objects are returned by the *SpaceNode.query_point_neighbors()* method. Properties are

- `hitbox` - an instance of *HitboxNode* which collided
- `body` - a *BodyNode* instance that owns the hitbox
- `point` - a *geometry.Vector* coords of the nearest point of collision
- `distance` - a *geometry.Vector* with a distance to the point of collision

2.11 `statistics` — Statistics module

2.11.1 `StatisticsManager` reference

Statistics manager is a reporting tool, surfacing basic metrics of kaa engine. To get the statistics manager instance use the *get_global_statistics_manager()* method.

```
class MyScene(Scene):

    def __init__(self):

        self.stats_manager = get_global_statistics_manager()


    def update(self, dt):

        self.stats_manager.push_value('custom statistic', random.gauss(10, 2))

        print(self.stats_manager.get_last_all())
        print(self.stats_manager.get_analysis_all())
```

class statistics.**StatisticsManager**

Instance methods:

StatisticsManager.**get_last_all**()
Returns the last value of each metric. Returned is a list of tuples in form of ('statistic name', value)

StatisticsManager.**get_analysis_all**()
Returns aggregated metric data. Returned is a list of tuples in form of ('statistic name', <StatisticAnalysis instance>). Check out *StatisticAnalysis* for more information.

StatisticsManager.**push_value**(*stat_name*, *value*)
Allows to push your own custom statistic. `stat_name` must be a string, and `value` must be a double. Your custom statistic will be reported by *get_last_all()* and *get_analysis_all()* methods.

2.11.2 StatisticAnalysis reference

class statistics.**StatisticAnalysis**
The `StatisticAnalysis` object wraps statistical properties of a larger sample of measurmenets.

Instance properties:

StatisticAnalysis.**samples_count**
Size of a sample.

StatisticAnalysis.**last_value**
The most recent value.

StatisticAnalysis.**mean_value**
The mean value.

StatisticAnalysis.**standard_deviation**
The standard deviation.

StatisticAnalysis.**max_value**
The maximum value.

StatisticAnalysis.**min_value**
The minimum value.

2.11.3 get_global_statistics_manager() reference

statistics.**get_global_statistics_manager**()
A method to get the *StatisticsManager* instance.

2.12 sprites — Using image assets

2.12.1 Sprite reference

class sprites.**Sprite**(*image_filepath*)

Sprite instance represents an image. The constructor accepts a path to a file. Supported formats are png and jpg.

Sprites instances are immutable.

If you want to load just a fragment of the image from a file, use the *Sprite.crop()* method.

If the file contains a spritesheet with multiple frames, use a helper function *split_spritesheet()* to automatically create a Sprite for each frame.

If the file includes multiple spritesheets, use the combination of *Sprite.crop()* and *split_spritesheet()* to 'cut' all frames from their respective areas.

For information how to draw a Sprite on the screen or how to create animations, *see here*.

A full example:

```
import os
from kaa.sprites import Sprite
from kaa.engine import Engine, Scene
from kaa.geometry import Vector
from kaa.nodes import Node

class MyScene(Scene):

    def __init__(self):
        self.root.add_child(Node(position=Vector(100,100),
                                 sprite=Sprite(os.path.join('demos', 'assets',
↪'python_small.png'))))

    def update(self, dt):

        for event in self.input.events():
            if event.system and event.system.quit:
                self.engine.quit()

with Engine(virtual_resolution=Vector(400, 200)) as engine:
    scene = MyScene()
    engine.window.size = Vector(400, 200)
    engine.window.center()

    engine.run(scene)
```

Instance Properties

Sprite.**size**

Returns Sprite size (width and height), as *geometry.Vector*

Sprite.**origin**

If the sprite was a result of a crop, it will return crop's origin point. Otherwise it'll return Vector(0,0)

Instance methods

Sprite.**crop**(*origin*, *dimensions*)

Returns a new Sprite, by cropping the original sprite.

The `origin` parameter is the start position of the crop - pass *`geometry.Vector`* indicating the (x,y) coordinates of the start position

The `dimensions` determines is the width and height of the crop - pass *`geometry.Vector`* where x and y are desired width and height respectively.

```
from kaa.sprites import Sprite
from kaa.geometry import Vector

# inside a Scene's __init__:
sprite = Sprite('path/to/sprite.png')  # sprite.png being a 1000x1000 px file.
print(sprite.size) # V[1000x1000]
new_sprite = sprite.crop(Vector(150,200), Vector(20,30))  # crop a new (20x30)␣
↪sprite, starting at (150,200)
print(new_sprite.size) # V[20,30]
```

2.12.2 `split_spritesheet()` reference

`sprites.`**`split_spritesheet`**(*spritesheet*, *frame_dimensions*, *frames_offset=0*, *frames_count=None*, *frame_padding=None*)

When an image file is a spritesheet you need to 'cut' it into individual Sprites (individual frames), which you can then use for making an animation using *`transitions.NodeSpriteTransition`*. This utility function does the cutting for you. It takes the following params:

- `spritesheet` - a *`Sprite`* instance holding your spritesheet
- `frame_dimensions` - dimensions of a single frame, expects *`geometry.Vector`* where x is frame width and y is frame height
- `frames_offset` - if you're interested in getting a subset of the frames, pass the start frame index. Default offset is zero (start from the first frame)
- `frames_count` - if you're interested in getting just a subset of the frames, pass the number of frames. By default the function will 'cut' as many frames as geometrically possible.
- `frame_padding` - some spritesheet tools can add a padding to each frame, if your spritesheet is using that feature pass a *`geometry.Vector`* where x is left+right padding and y is top+bottom padding. Example: if using 1-pixel padding on all sides, pass Vector(2,2)

The function will process the spritesheet going from left to right and from top to bottom, cutting out the individual frames, returning a list of Sprites.

```
# suppose a spritesheet.png is a 1000x1000 file with a hundred frames of 100x100␣
↪size
spritesheet = Sprite('path/to/spritesheet.png')
# cut all frames:
all_frames = split_spritesheet(spritesheet, Vector(100, 100))
# cut 10 frames, from 20 to 29
subset_of_frames = split_spritesheet(spritesheet, Vector(100, 100), frames_
↪offset=20, frames_count=10)
# crop a 40x40 area starting from (20,20), and cut five frames starting from␣
↪frame 3
another_subset_of_frames = split_spritesheet(spritesheet.crop(Vector(20,20),␣
↪Vector(40,40)),
    frame_offset=3, frames_count=5)
```

2.13 timers — a simple timer

2.13.1 Timer reference

class timers.**Timer**(*callback_func*)

Timer will call the callback_func callable after time specified when calling the *start()* or *start_global()* method.

The callback_func callable implementation receives one parameter - timer_context which is a TimerContext instance.

```
def my_func(self, timer_context):
    print('Triggered by the timer!')

global_timer = Timer(my_func)
global_timer.start_global(1.5)  # in seconds
```

```
# ... somewhere inside a Scene:

def my_func(self, timer_context):
    print('Triggered by the timer!')

def add_timer(self):
    timer = Timer(my_func)
    timer.start(1.5, scene=self)  # in seconds
```

The callback_func may return a numeric value. It will reset the timer, allowing to run it in a loop:

```
def my_func(self, timer_context):
    new_interval = random.uniform(1.0, 2.0)
    print('Resetting the timer with interval of {} seconds'.format(new_interval))
    return new_interval  # this resets the timer

global_timer = Timer(my_func)
global_timer.start_global(1.5)  # in seconds
```

Instance properties:

Timer.**is_running**

Returns True if the timer is running.

Instance methods:

Timer.**start**(*interval*, *scene*)

Starts the timer in a context of a specific *engine.Scene* instance. After interval seconds, the timer's callback function (defined in the constructor) will be invoked.

There are few reasons why you may want a scene instance associated with a timer:

- If you change scene to a new one, the timers associated with the previous scene will stop running automatically
- When a scene gets destroyed, timers associated with that scene will be destroyed as well and you won't receive any surprise callbacks.
- Timers utilize *engine.Scene.time_scale* property.

Calling *start()* on a running timer resets the timer.

`Timer.`**`start_global`**`(`*interval*`)`
Same as *start()* but does not require passing a scene. Use it if you need a "global" timer.

`Timer.`**`stop`**`()`
Stops the timer.

2.13.2 `TimerContext` reference

An object passed to timer's callback function

Instance properties:

`TimerContext.`**`scene`**
Read only. Gets the scene (as *engine.Scene* instance). Will be None if timer was called with *Timer.start_global()* method.

`TimerContext.`**`interval`**
Read only. Interval with which the timer was called.

2.14 `transitions` — A quick and easy way to automate transforming your nodes

When writing a game you'll often want to apply a set of known transformations to a *nodes.Node*. For example, you want your Node to move 100 pixels to the right, then wait 3 seconds and return 100 pixels to the left. Or you'll want the node to pulsate (smoothly change its scale between some min and max values), or rotate back and forth. YOu may want to have any combination of those effects applied (either one after another or parallel). There's an unlimited number of such visual transformations and their combinations, that you may want to have in your game.

You can of course implement all this, by having a set of boolean flags, time trackers, etc. and use all those helper variables to manually change the desired properties of your nodes from within the update() method. But there is an easier way: the mechanism is called Transitions. A single Transition object is a recipe of how a given property of a Node (position, scale, rotation, color, sprite, etc.) should change over time. Transition can be played once, given number of times or in a loop. You can also chains transitions to run one after another or in parallel.

Transitions are the primary way of creating animations. Since animation is nothing else than just changing Node's sprite over time, the transition mechanism comes useful for that purpose.

2.14.1 Common transition parameters

Note: All transitions are immutable.

To create a Transition you'll typically need to pass the following parameters:

- `advance_value` - advance value for given transition type (e.g. target position for *NodePositionTransition*).
- `duration` - transition duration time, in seconds
- **`advance_method` - an enum value of *`AttributeTransitionMethod`* type which determines how the `advance_va`**

  - `AttributeTransitionMethod.set` - node's property will be changed towards the advance_value over time
  - `AttributeTransitionMethod.add` - node's property will be changed towards the current value + advance_value over time

  - `AttributeTransitionMethod.multiply` - node's property will be changed towards the current value * advance_value over time

- `loops` - Optional. How many times the transition should "play". Set to 0 to play infinite number of times. Default is 1.
- `back_and_forth` - Optional. If set to `True`, the transition will be played back and forth (that counts as one "loop"). Default is False.
- `easing` - Optional. An enum value of *`easings.Easing`* - specifies the rate of change of a value over time. Default is Easing.none which really means a linear easing.

Note: The `duration` parameter always refers to one loop, one direction. So for example, transition with the following set of parameters: `duration=1., loops=3, back_and_forth=True` will take 6 seconds. 1 second played back and forth is 2 seconds, and it will be played 3 times, hence a total time of 6 seconds.

Note: If `back_and_forth` is set to `True`, the transition will play back and forth which counts as one loop.

All transitions use linear easing. More built-in easing types are to be added soon.

2.14.2 Examples

Change position of a node, from (100,100) to (30, 70) over 2 seconds.

```
node = Node(position=Vector(100, 100), sprite=Sprite('image.png'))
node.transition = NodePositionTransition(Vector(30, 70), 2.)
```

Change position of a node, from (100,100) by (30, 70), i.e. to (130, 170) over 2 seconds.

```
node = Node(position=Vector(100, 100), sprite=Sprite('image.png'))
node.transition = NodePositionTransition(Vector(30, 70), 2., advance_
↪method=AttributeTransitionMethod.add)
```

Change position of a node, from (100, 100) by (x30, x70), i.e. to (3000, 7000) over 2 seconds.

```
node = Node(position=Vector(100, 100), sprite=Sprite('image.png'))
node.transition = NodePositionTransition(Vector(30, 70), 2., advance_
↪method=AttributeTransitionMethod.multiply)
```

Change position of a node, from (100,100) to (30, 70) then back to the initial position (100,100) over 2 seconds.

```
node = Node(position=Vector(100, 100), sprite=Sprite('image.png'))
node.transition = NodePositionTransition(Vector(30, 70), 2., back_and_forth=True)
```

Change position of a node, from (100,100) to (30, 70) then get back to the initial position over 2 seconds. Repeat it 3 times.

```
node = Node(position=Vector(100, 100), sprite=Sprite('image.png'))
node.transition = NodePositionTransition(Vector(30, 70), 2., loops=3, back_and_
↪forth=True)
```

Change the scale of a node (twice on the X axis and three times on the Y axis) over 1 second.

```
node = Node(position=Vector(100, 100), sprite=Sprite('image.png'))
node.transition = NodeScaleTransition(Vector(2, 3), 1.)
```

Change the scale of a node (twice on the X axis and three times on the Y axis) over 1 second. Repeat indefinitely (creating pulsation effect).

```
node = Node(position=Vector(100, 100), sprite=Sprite('image.png'))
node.transition = NodeScaleTransition(Vector(2, 3), 1., loops=0)
```

Rotate the node 90 degrees clockwise over 3 seconds

```
node = Node(position=Vector(100, 100), sprite=Sprite('image.png'))
node.transition = NodeRotationTransition(math.pi/2, 3.)
```

Change position of a node by (150, 100) over 2 seconds, then enlarge it twice over 1 second, then do nothing for 2 seconds, finally rotate it 180 degrees over 3 seconds. Play the whole sequence two times, back and forth.

```
node = Node(position=Vector(100, 100), sprite=Sprite('image.png'))
transitions = [
    NodePositionTransition(Vector(150, 100), 2., advance_
↪method=AttributeTransitionMethod.add),
    NodeScaleTransition(Vector(2, 2), 1.),
    NodeTransitionDelay(2.),
    NodeRotationTransition(math.pi, 3.)
]
node.transition = NodeTransitionsSequence(transitions, loops=2, back_and_forth=True)
```

Do everything the same like in previous example but have the node *simultaneously* change its color to red, back and forth in 1500 milisecond intervals.

```
node = Node(position=Vector(100, 100), sprite=Sprite('image.png'))
transitions = [
    NodePositionTransition(Vector(150, 100), 2., advance_
↪method=AttributeTransitionMethod.add),
    NodeScaleTransition(Vector(2, 2), 1.),
    NodeTransitionDelay(2.),
    NodeRotationTransition(math.pi, 3.)
]
color_transition = NodeColorTransition(Color(1,0,0,1), 1.5, loops=0, back_and_
↪forth=True)

node.transition = NodeTransitionsParalel([
    color_transition,
    NodeTransitionsSequence(transitions, loops=2, back_and_forth=True)
])
```

Change position of a node, from (100,100) to (30, 70) over 2 seconds and call function my_func when the transition ends.

```
def my_func(transitioning_node):
    print('Function called!')

node = Node(position=Vector(100, 100), sprite=Sprite('image.png'))
node.transition = NodeTransitionSequence([
    NodePositionTransition(Vector(30, 70), 2.),
    NodeTransitionCallback(my_func)])
```

Change sprite of a node, creating an animation effect:

```
spritesheet = Sprite(os.path.join('assets', 'gfx', 'spritesheet.png')
frames = split_spritesheet(spritesheet, Vector(100,100)) # cut the spritesheet into
↪<Sprite> instances
animation = NodeSpriteTransition(frames, duration=2., loops=0, back_and_forth=False)
node = Node(position=Vector(100, 100), transition=animation)
```

Change z_index of a node over time:

```
node = Node(position=Vector(100, 100), sprite=Sprite('image.png'))
node.transition = NodeZIndexSteppingTransition([1,2,3,4,5,6,10,100], 1000)
```

2.14.3 `NodePositionTransition` reference

class transitions.**NodePositionTransition**(*advance_value*, *duration*, *advance_method=AttributeTransitionMethod.set*, *loops=1*, *back_and_forth=False*, *easing=Easing.none*)

Use this transition to change Node's position gradually over time, towards given advance_value or by given advance_value.

The `advance_value` param must be a *`geometry.Vector`* and is the target position value (or position change value)

Refer to the *Common transition parameters* and *Examples* sections for information on other parameters used by the transition.

2.14.4 `NodeRotationTransition` reference

class transitions.**NodeRotationTransition**(*advance_value*, *duration*, *advance_method=AttributeTransitionMethod.set*, *loops=1*, *back_and_forth=False*, *easing=Easing.none*)

Use this transition to change Node's rotation gradually over time, towards given advance_value or by given advance_value.

The `advance_value` param must be a float and is the target rotation value (or rotation change value), *in radians*.

Refer to the *Common transition parameters* and *Examples* sections for information on other parameters used by the transition.

2.14.5 `NodeScaleTransition` reference

class transitions.**NodeScaleTransition**(*value*, *duration*, *advance_method=AttributeTransitionMethod.set*, *loops=1*, *back_and_forth=False*, *easing=Easing.none*)

Use this transition to change Node's scale gradually over time, towards given advance_value or by given advance_value.

The `advance_value` param must be a *`geometry.Vector`* and is the target scale value (or scale change value) for X and Y axis respectively.

Refer to the *Common transition parameters* and *Examples* sections for information on other parameters used by the transition.

2.14.6 `NodeColorTransition` reference

class transitions.**NodeColorTransition**(*value*, *duration*, *advance_method=AttributeTransitionMethod.set*, *loops=1*, *back_and_forth=False*, *easing=Easing.none*)

Use this transition to change Node's scale gradually over time, towards given advance_value or by given advance_value.

The `advance_value` param must be a *`colors.Color`* and is the target color value (or color change value).

Note that each component of the color (R, G, B, and A) is trimmed to a 0-1 range, so when using `advance_method=AttributeTransitionMethod.set` or `advance_method=AttributeTransitionMethod.multiply` which would result in R G B or A going above 1 or below 0 - the value will be capped at 1 and 0 respectively.

Refer to the *Common transition parameters* and *Examples* sections for information on other parameters used by the transition.

2.14.7 `BodyNodeVelocityTransition` reference

class transitions.**BodyNodeVelocityTransition**(*value*, *duration*, *advance_method=AttributeTransitionMethod.set*, *loops=1*, *back_and_forth=False*, *easing=Easing.none*)

Use this transition to change BodyNode's velocity gradually over time, towards given advance_value or by given advance_value.

The `advance_value` param must be a *`geometry.Vector`* and is the target velocity value (or velocity change value).

Refer to the *Common transition parameters* and *Examples* sections for information on other parameters used by the transition.

2.14.8 `BodyNodeAngularVelocityTransition` reference

class transitions.**BodyNodeAngularVelocityTransition**(*value*, *duration*, *advance_method=AttributeTransitionMethod.set*, *loops=1*, *back_and_forth=False*, *easing=Easing.none*)

Use this transition to change BodyNode's angular velocity gradually over time, towards given advance_value or by given advance_value.

The `advance_value` param must be a number and is the target angular velocity value (or angular velocity change value), *in radians*

Refer to the *Common transition parameters* and *Examples* sections for information on other parameters used by the transition.

2.14.9 `NodeSpriteTransition` reference

class transitions.**NodeSpriteTransition**(*sprites*, *duration*, *loops=1*, *back_and_forth=False*, *easing=Easing.none*)

Use this transition to create animations. The transition will change Node's sprite over time specified by the `duration` parameter, iterating through sprites list specified by the `sprites` parameter.

The `sprites` must be an iterable holding *`sprites.Sprite`* instances. To cut a spritesheet file into individual sprites (individual frames) use the utility function *`sprites.split_spritesheet()`*

The `loops` and `back_and_forth` parameters work normally - refer to the *Common transition parameters* section for more information on those parameters.

2.14.10 `NodeZIndexSteppingTransition` reference

class transitions.**NodeZIndexSteppingTransition**(*z_index_list*, *duration*, *loops=1*, *back_and_forth=False*, *easing=Easing.none*)

Allows to change z_index of a node over time.

The `z_index_list` must be an iterable with z_index values.

2.14.11 `NodeTransitionsSequence` reference

class transitions.**NodeTransitionSequence**(*transitions*, *loops=1*, *back_and_forth=False*)

A wrapping container used to chain transitions into a sequence. The sequence will run one transition at a time, next one being executed when the previous one finishes.

The `transitions` parameter is an iterable of transitions.

The iterable can include a list of 'atomic' transitions such as *`NodePositionTransition`*, *`NodeScaleTransition`*, *`NodeColorTransition`* etc. as well as other *`NodeTransitionSequence`*, or *`NodeTransitionsParallel`* thus building a more complex structure.

The loops and back_and_forth parameters work normally, but are applied to the whole sequence.

See the *Examples* sections for a sample code using NodeTransitionSequence.

2.14.12 `NodeTransitionsParallel` reference

class transitions.**NodeTransitionsParallel**(*transitions*, *loops=1*, *back_and_forth=False*)

A wrapping container used to make transitions run in parallel.

The `transitions` parameter is an iterable of transitions which will be executed simultaneously.

The iterable can include a list of 'atomic' transitions such as *`NodePositionTransition`*, *`NodeScaleTransition`*, *`NodeColorTransition`* etc. as well as other *`NodeTransitionSequence`*, or *`NodeTransitionsParallel`* thus building a more complex structure.

You may have two contradictory transitions running in parallel, for example two *`NodePositionTransition`* trying to change node position in opposite directions. Contrary to intuition, they won't cancel out (regardless of advance_method being `add` or `set`). If there are two or more transitions of the same type running in paralel, then the one which is later in the list will be used and all the preceding ones will be ignored.

Since transitions runing in parallel may have different durations, the `loops` parameter is using the following logic: The longest duration is considered the "base" duration. Transitions whose duration is shorter than the base duration will wait (doing nothing) when they complete, until the one with the "base" duration ends. When the "base" transition ends, the new loop begins and all transitions start running in parallel again.

The `back_and_forth=True` is using the same logic: the engine will wait for the longest transition to end before playing all parallel transitions backwards.

See the *Examples* sections for a sample code using NodeTransitionsParallel.

Like all other transitions, NodeTransitionsParallel is immutable. That causes problems when you want transitions to be managed independently. Consider a situation where you want to have a Node with sprite animation (NodeSpriteTransition) and some other transition (e.g. NodePositionTransition), both running simuntaneously. Suppose you do that by wrapping the two transitions in *NodeTransitionsParallel*. Now, if you want to change just the sprite animation transition **without changing the state of the position transition** (a perfectly valid case in many 2D games), you won't be able to do that because NodeTransitionsParallel is immutable!

To solve that problem, you should use `NodeTransitionsManager` - it allows running and managing multiple simultaneous transitions on a Node truly independently from each other.

2.14.13 `NodeTransitionDelay` reference

class transitions.**NodeTransitionDelay**(*duration*)

Use this transition to create a delay between transitions in a sequence.

The `duration` parameter is a number of seconds.

See the *Examples* sections for more information.

2.14.14 `NodeTransitionCallback` reference

class transitions.**NodeTransitionCallback**(*callback_func*)

Use this transition to get your own function called at a specific moment in a transitions sequence. A typical use case is to find out that a transition has ended.

The `callback_func` must be a callable.

See the *Examples* sections for a sample code using NodeTransitionCallback

2.14.15 `NodeCustomTransition` reference

class transitions.**NodeCustomTransition**(*prepare_func*, *evaluate_func*, *duration*, *loops=1*, *back_and_forth=False*, *easing=Easing.none*)

Use this class to write your own transition.

`prepare_func` must be a callable. It will be called once, before the transition is played. It receives one parameter - a node. It can return any value, which will later be used as input to `evaluate_func`

`evaluate_func` must be a callable. It will be called on each frame and it's the place where you should implement the transition logic. It will receive three parameters: `state`, `node` and `t`. The `state` is a value you have returned in the `prepare_func` callable. The `node` is a node which is transitioning. The `t` parameter is a value between 0 and 1 which indicates transition time duration progress.

The `loops` and `back_and_forth` paramters behave normally - see the *Common transition parameters* section.

```
custom_transition = NodeCustomTransition(
        lambda node: {'positions': [
            Vector(random.uniform(-100, 100), random.uniform(-100, 100))
            for _ in range(10)
        ]},
        lambda state, node, t: setattr(
            node, 'position',
            state['positions'][min(int(t * 10), 9)],
```

(continues on next page)

(continued from previous page)

```
        ),
        10.,
        loops=5,
    )
```

2.14.16 `NodeTransitionsManager` reference

class transitions.**NodeTransitionsManager**

Node Transitions Manager is accessed by the transitions_manager property on a *nodes.Node*. It allows to run multiple transitions on a node at the same time. Unlike *NodeTransitionsParallel*, which also runs multiple transitions simultaneously, the transitions managed by the NodeTransitionsManager are truly isolated. It means you can manage them (stop or replace them) **individually** not affecting other running transitions. This is not possible with transitions inside *NodeTransitionsParallel*, because the wrapper is immutable.

The manager offers a simple dictionary-like interface with two methods: *get()* and *set()* to access and set transitions by a string key.

Note that the transition manager is used when you set transition on a Node via the *transition property*. That transition can be accessed via `get('__default__')`

Similarly to *NodeTransitionsParallel* when you set two contradictory transitions of the same type to run on the manager (for example position transitions that pull the node in two opposite direction) - they will not cancel out. One of them will 'dominate'. It is undetermined which one will dominate therefore it's recommended not to compose transitions that way (why would you want to do it anyway?).

NodeTransitionsManager.**get**(*transition_name*)

Gets a transition by name (a string).

`Node.transitions_manager.get('__default__')` is an equivalent of *Node.transition* getter.

NodeTransitionsManager.**set**(*transition_name*, *transition*)

Sets a transition with a specific name (a string). The `transition` object can be any transition, either 'atomic' or a serial / parallel combo.

`Node.transitions_manager.set('__default__', transition)` is an equivalent of *Node.transition* setter.

```
node = Node(position=Vector(15, 60))
node.transitions_manager.set('my_transition', NodePositionTransition(Vector(100,
↪100), duration=0.300, loops=0))
node.transitions_manager.set('other_transition', NodeRotationTransition(math.pi/
↪2))
node.transitions_manager.set('can_use_sequence_coz_why_not', ␣
↪NodeTransitionsSequence([
    NodeScaleTransition(Vector(2, 2), 1.),
    NodeTransitionDelay(2.),
    NodeColorTransition(Color(0.5, 1, 0, 1), 3.)],
    loops=2, back_and_forth=True))
```

2.14.17 `AttributeTransitionMethod` reference

class transitions.**AttributeTransitionMethod**

Enum type used to identify value advance method when using transitions

Available values are:

- AttributeTransitionMethod.set
- AttributeTransitionMethod.add
- AttributeTransitionMethod.multiply

2.15 All kaa imports cheat sheet

```
from kaa.audio import Sound, SoundPlayback, Music, AudioStatus

from kaa.colors import Color

from kaa.easings import Easing, ease, ease_between

from kaa.engine import Engine, Scene, VirtualResolutionMode, get_engine

from kaa.fonts import Font, TextNode

from kaa.geometry import Vector, Segment, Circle, Polygon, PolygonType, Alignment,␣
↪Transformation, BoundingBox, classify_polygon

from kaa.input import Event, Keycode, MouseButton, ControllerButton, ControllerAxis,␣
↪CompoundControllerAxis

from kaa.log import get_core_logging_level, set_core_logging_level, CoreLogLevel,␣
↪CoreHandler,

from kaa.nodes import Node

from kaa.physics import SpaceNode, BodyNode, HitboxNode, BodyNodeType, CollisionPhase

from kaa.renderer import Renderer

from kaa.statistics import get_global_statistics_manager, StatisticsManager,␣
↪StatisticAnalysis

from kaa.sprites import Sprite, split_spritesheet

from kaa.timers import Timer

from kaa.transitions import NodeTransitionsSequence, NodeTransitionsParallel,␣
↪NodeCustomTransition,
    AttributeTransitionMethod, NodePositionTransition, NodeRotationTransition,␣
↪NodeScaleTransition,
    NodeColorTransition, BodyNodeVelocityTransition,␣
↪BodyNodeAngularVelocityTransition, NodeTransitionDelay,
    NodeTransitionCallback, NodeSpriteTransition, NodeZIndexSteppingTransition
```

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z