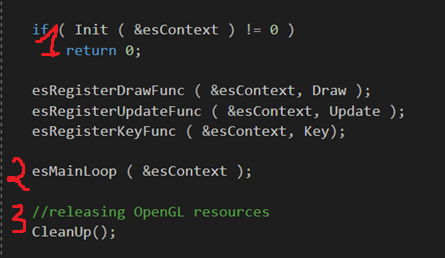
# SoyEngine

Overview

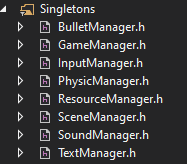


1. Init() : Initialize the game, loading scene and necessary files
2. esMainLoop() : Game Loop
3. CleanUp() : Clean Game trashes

esMainLoop:

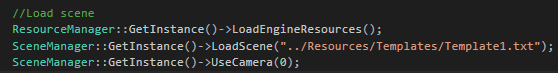
* Key(): Only get called when a key is pressed or released
* Update(): Is called every frames
* Draw(): Is called every frames as well but should be implied as a container for rendering sprites

## Singletons

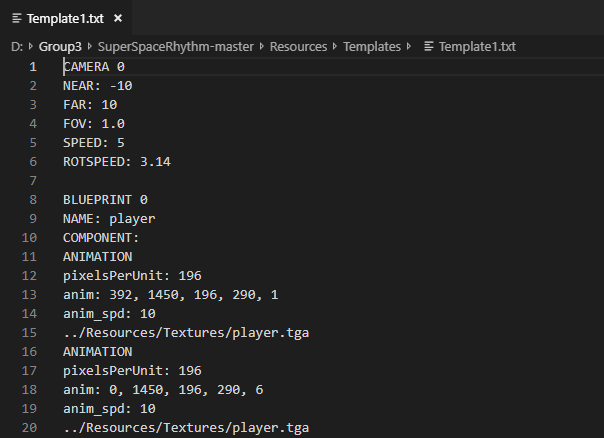


* BulletManager: Ask Hai
* GameManager: Manage how the enemy spawn, when to drop the boss. Basically a dungeon master
* InputManager: Store Input and allow key binds
* PhysicManager: Manage physics with box2d
* ResourceManager: Holds all loaded resources informations.
* SceneManager: Holds all GameObjects in the current scene.
  + Call Update() and Draw() for all GameObjects.
* SoundManager: Manage Sounds
* TextManager: Allow you to create text object and render them

## Loading a scene



When starting the game, Init() will load a template file with SceneManager::GetInstance()->LoadScene(char \* scenefilename);



*Here is an example of a template file*

Template file will contain 2 object format: **camera** & **blueprint**

A **camera** contains details about the camera object that is going to be used

A **blueprint** is self-explainable

* It contains information to be used when spawning a GameObject (GO) with Spawn(Blueprint\* blueprint, ...) in GameManager
* You can use SceneManager::GetInstance()->GetBlueprintByName(char \* name); and it will return the first Blueprint \* which has that name.

BLUEPRINT # //Note that currently # has no usage

NAME: BLUEPRINT\_NAME

COMPONENT:

COMPONENT\_NAME

COMPONENT\_DETAILS

[COMPONENT\_NAME]

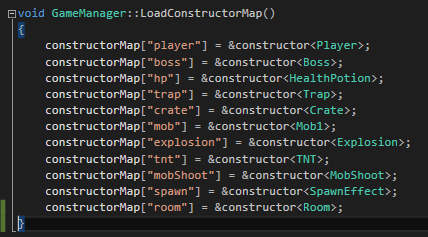
[COMPONENT\_DETAILS]

## Spawning a GO

Every GO that is spawned into/removed from the game is within GameManager, SceneManager should only act as an intermediate for managing memory. To spawn a GO, use Spawn() function:

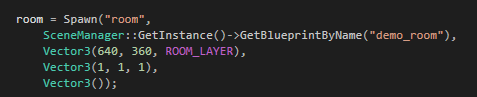


Before you spawn a GO, be sure that that object has its’ constructor included in LoadConstructor()

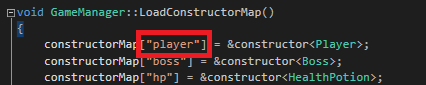


*Note that Player, Boss and all classes in the Angle Brackets < > must be a GameObject or a child of it*

Now this is how to spawn a GO:



The first parameter *constructorName* refers to the name you gave to the constructor

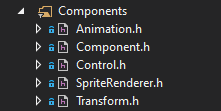
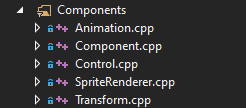


The second parameter blueprint is the pointer to the blueprint that you wish to use to create a GO

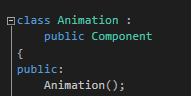
The rest postion, scale, rotation is self-explainatory.

## How to add a component?

A **Blueprint** is a list component. To add a custom component to a blueprint in Template file you must create one first.

First thing first, make sure your new component inherit the Component class



Every components has 3 virtual functions that can be overrided: Update(), Draw() and Clone()

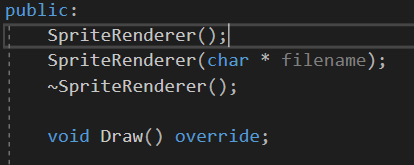


Update() is called every frames by SceneManager::GetInstance()->Update() inside Opengl’s Update()

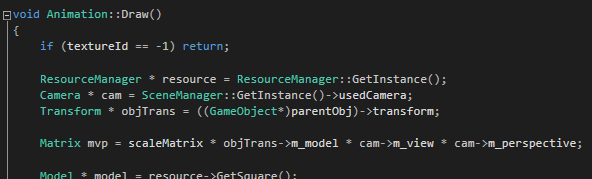
Draw() is called every frames by SceneManager::GetInstance()->DrawAll() inside Opengl’s Draw()

Clone() is used by SceneManager::GetInstance()->SpawnObject() to duplicate Components

To override them, simply make a function with the same name in the child class of Component class. The override keyword is to remind the compiler that this is an override function.



Then write its’ definition in the .cpp file



Please keep in mind that defining the Clone() method is **mandatory** for all Components, otherwise GameManager can not spawn your object.

At this point, your component can be added during runtime and should operate like how you expect it to. However, in order for your component to be recognized by the SceneManager::GetInstance()->LoadScene() function, you must append some code to it. Go to SceneManager’s GetComponent()



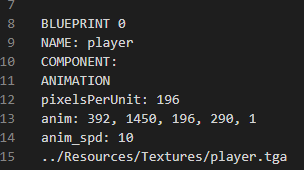
Simply add:

else if (!line.compare(“[COMPONENT NAME]”)) {

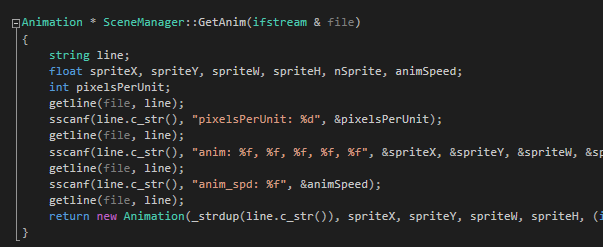
blueprint->AddComponent(GetFunc(file));

}

[COMPONENT NAME] should be the name that will be used to recognize the component in the Template file.

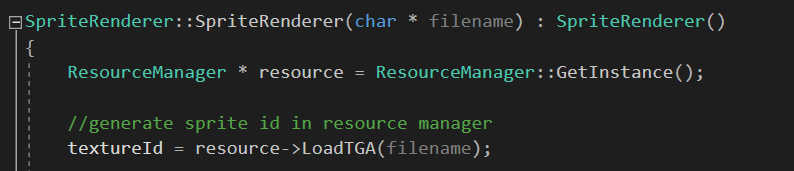


Then create a new function for reading component information: GetFunc(file) = Get+[component shortened name](file).

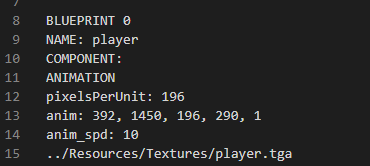


* 1. string line; //To read file
  2. <Define variables for the component>
  3. Add codes on how the component informations should be read from file (getline(file, line) for next line, sscanf(line.c\_str(), “[format]”, &variable) to get variables). Take a look at how the currently implemented component are written to get a clearer idea.
  4. Return new Component object

If you wish to use a custom constructor for your component, it’s good practice to inherit it from the default constructor

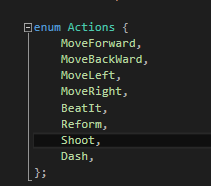


You should be able to recognize your component from the scene file now

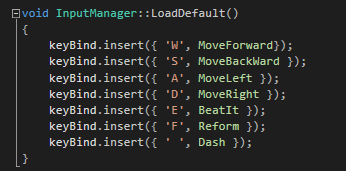


## Detecting Input

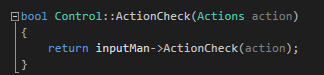
Every key presses, every mouse click (except RMB cause we are using opengl es) is registered in InputManager. To add a key to detection, first create a corresponding action in InputManager.h:



Then assign a key (current input manager does not handle key combination like Ctrl+C) to that action in InputManager.cpp’s LoadDefault():



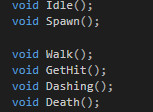
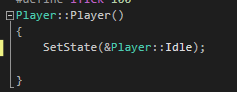
Use InputManager::ActionCheck(Action) to know if that action is true or not. Here is an example in Control Component.



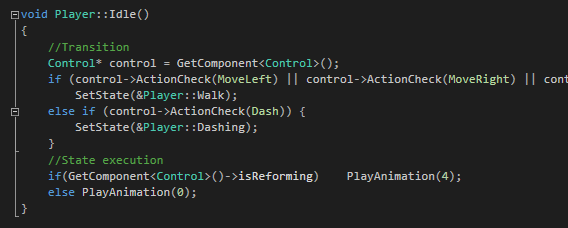
That’s it!

## StateMachine

Let take a look at the Player class. The default state is Idle()



These are the state function, they define how the player will act in those state.



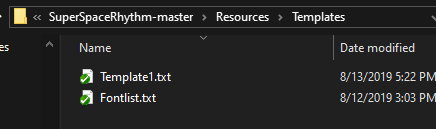
At the start, define all possible transition to other states using SetState().

Then insert the code of whatever you want the Player to do in that state. Please note that this is done everyframe, so if something isn’t meant to be repeated (like activating the first frame of an animation) should have a boolean to prevent undesirable result.

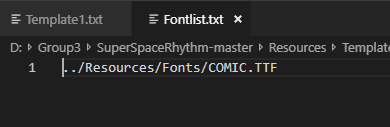
Do so for every states you want, disable control for a realistic feel and you are done!

## TextManager

Open Fontlist.txt



Add new font



Use TextManager::GetInstance()->AddText(...);

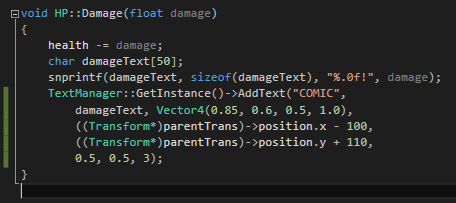
First parameter is font name minus the file extension.

Second parameter is the context

Color, positionX, positionY

ScaleX, ScaleY

And TimeToLive (in second) // You can omit the TTL and the text will be permanent, just remember to use TextManager::GetInstance()->DeleteText(...);



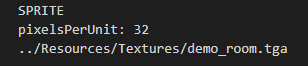
Render Text 101 is finished.

## Appendix

Contains information such as Component usage, etc

SpriteRenderer

*Render one sprite*

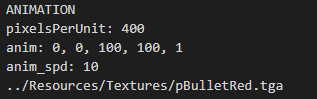


Sprite size (in pixel) / pixelPerUnit = Sprite size (in grid, default grid size is 40.0f pixel)

Then the link to the sprite file

### Animation

*Render Animation*



Sprite size (in pixel) / pixelPerUnit = Sprite size (in grid, default grid size is 40.0f pixel)

Anim: startX, startY, spriteW, spriteH, numberOfSprites

//startX, startY: the position of the top left point of the first animation

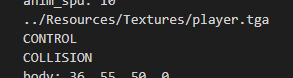
//spriteW, spriteH: sprite width, height

//anim\_spd: frames per sprite

Then the link to the sprite sheet

### Control

Just add it and you can control it

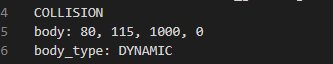


### HP (Hit points)



Health: health amount

### Collision2D



Body: colW, colH, mass, res

//colW, colH: collision width, height

//mass: mass

//res: resitution

//body\_type: box2D body type (STATIC, KINEMATIC, DYNAMIC)