**2D Engine – A C++ & SDL2 Application**

Donovan Lay

CSCI 437 Intro To 3D Game Graphics

This is meant to serve as a short tutorial to understand how to make your own game in the engine and run it. There will also be another documentation page that goes over the functions and how they all work. To compile your engine you will need both SDL2 and SDL\_Image downloaded, and make sure you add the include and library paths.

**Making your own game**

In order to make your own game you’re going to want to open up the project folder, then go \Games\ and create a new folder inside of that for whatever you want to name your game. Then, inside of that folder I recommend having 2 more folders, one for assets, and one for your source code. In your source code you’re going to want one, and only one, class that inherits from the Game class, and then you can also create as many Sprite subclasses as you need. Make sure when you create these subclasses that your include paths are correct, for instance in my Ship class, a Sprite subclass in the Space Game, I have



Next you need to program in your mechanics, including user input, and specifically overwriting the methods named Update, PlayerInput, DefaultBehavior, CollisionBehavior, and DealDamage for the Sprite class, and InitGame from the Game class.

**User Input**

Some user input is already handled by the program, namely pressing P to pause the game, T to toggle the mouse cursor, Q or Esc to quit. H is also built in to hide the window, unfortunately there is currently a bug with this as H does not toggle the window back. Other accepted inputs that get sent to PlayerInput are the keys W, A, S, and D, and the left mouse button. This should be enough for a simple 2D game, if you wanted to add more you definitely could, and it should be relatively straight forward to do so just by comparing other code to the key you want to add.

Now, to actually add your input from the preset buttons is fairly easy, I’ll show an example here: 

SDL has a few different ways to handle inputs, the method I chose was dealing with what it calls Events. The upside to Events, is as you can see, it should be fairly straightforward to understand, we’re checking the event type, and then based on that we’re just checking which key is which, ex. SDLK\_a is the A key. Under these blocks you put your logic. I left extra examples here commented out. First under the W key, the commented out code would allow the user to move up on the screen. Then there’s the A key, which currently, in tandem with the D key, the uncommented code only allows for the user to move left and right, sort of like an arcade style game. You could also comment that code out and swap it for the currently commented code, which would allow the user to rotate the ship, and then uncommenting the code under the W key except the MoveAngle, you’d have full 360-degree movement, and be able to move anywhere on the screen.

Back to Events, the downside is that despite the appropriate syntax, there doesn’t seem to be a way with Events to handle multiple inputs at once. The solution would be to change to another form of user input that has a type of Uint8\*. Unfortunately by time this was realized it would have taken too much time to fix, and there were bigger issues to deal with.

**How collisions detection works**

The approach to collision handling I chose is similar to the one Andy talked about in class, and it’s referred to as the Separating Axis Theorem, and the best way to imagine it is this: Imagine you have two shapes at some points in a room, their rotations do not matter. Now you directly face one of the faces of either shape and shine a light, if from this light setup, the shadows intersect, then it appears from here, that there is a collision. Now you do this for each face of each shape. If there is even one gap in-between the shadows of any side, then there is no possible collision, and that is proven by the theorem. If from every face there are no gaps, then there must be a collision.

Now this probably sounds overly complicated, but the reason for choosing this way are for a few different reasons. The first reason is that we avoid axis-aligned boxes, and the good part about this is that if we rotate a sprite that is rectangular in shape, we don’t have to worry about any stretching effect, the ‘collision box’ rotates with the sprite. Another reason is that the separate axis theorem works for any convex shape, so your collision box could be a triangle, a rectangle, a square, a circle, a hexagon, etc. As long as the shape doesn’t turn in on itself, this theorem applies.

Now, obviously I’m not ray-casting or shining a light everywhere, that was just an analogy and realistically it’s much more math based. First, you need to calculate the vertices of each shape, which is done automatically for you when you add a sprite into the engine. Now unfortunately, the only collision shape currently built in is a rectangle/square. The idea being in the future I would add in different types, and then the user would have to input the collision type when creating the sprite, which would the calculate vertices for a triangle, for example. Now with the vertices, which are points, we can treat them as vectors, as that’s all points really are, and use neighboring vertices to calculate the normalized normal vectors for each face of the each sprite, which we treat as a new, separate axis (now you see where the name comes from). We use the new axis, and project each vertex from each sprite down onto this axis and check for overlaps, and we do that for each axis, stopping if there is a gap. This theorem is also scalable, as it can scale into 3D space.

This method is actually very good and relatively quick if compensated for correctly. This method is considered a ‘narrow’ phase collision detection method. Usually collision detection is separated into two parts, the broad phase and the narrow phase. The broad phase uses different methods, one method is to make a Binary Space Partitioning Tree, and check which objects could *possibly* collide. Using something like this would save on performance because you aren’t always checking for collisions, you only check if collisions are possible. My engine currently does not have a broad phase, so I do believe there is a performance hit due to constantly doing narrow phase checks.

**Running your game**

Once you have programmed your game, you need to be able to run it. The approach to this is pretty simple: 1) Create a Scene 2) Create your custom Game object, and pass a pointer of the Scene to the constructor 3) Call InitGame on your game and pass the name of your game to it. That’s it! Do that and you can run your own custom game in the engine!