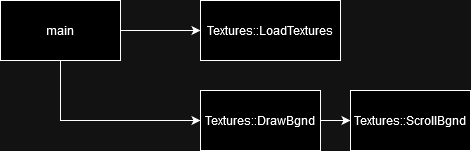
# Side Scrolling Shmup Design Document

### Understanding the template

In order to better understand the codebase i created a flowchart and procedure dependency for diagram for the template codebase. They are very simple for this template

#### Procedure Dependency Diagram



#### Flowchart



## Upgrading the template

In order to help me plan the changes i wanna make i made a flowchart for the program after the planned upgrades

I dont think the current system of storing the sprites in the template is easily extendable, and as such i will be building my own system, storing the sprites within an entity object.

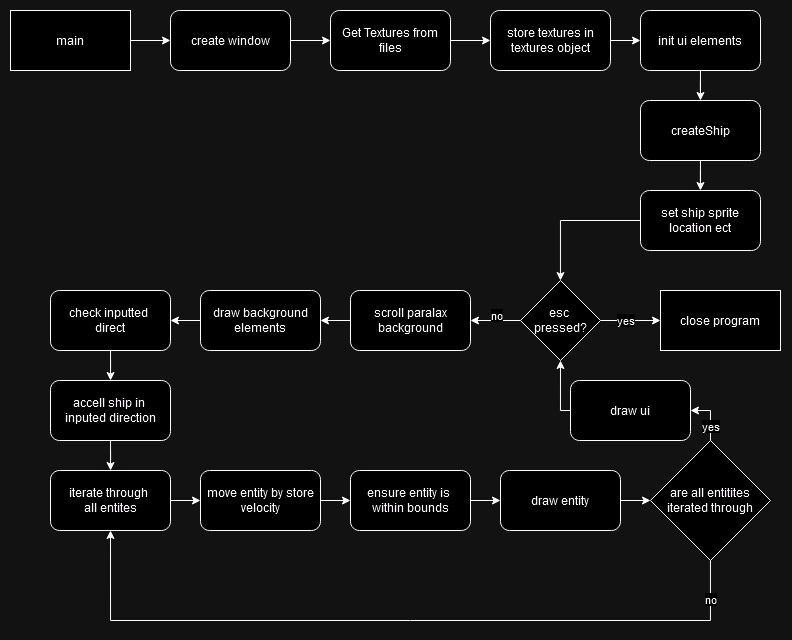
##### Entity object

At the core of the planned upgrades is the entity object, storing all the infomation and methords required to move the ship and any future objects required in the future. Building this as an entity object will allow the code to be neatly contained and minimise repeat/redundant code. For future proofing and as good practice i will then extend a ship class from that entity class in minimise the overhead and clutter for non ship objects that may be added in the future.

Here is some psudo code for what i predict will be contained within it

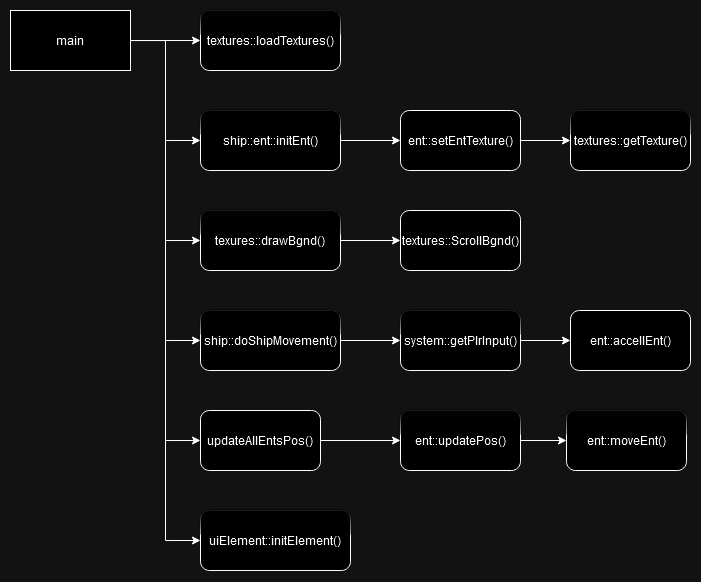
|  |
| --- |
| Entity object{  Float xPos, yPos  Sprite entSpr  initEnt{  setTexture()  setPos()  }  RenderIn{}  setPos{}  accellEnt{}  mvEnt{  clampToScreen()  }  updateEntPos{}  }  Ship::entity object{  handleUserInputs{  Vector2 inputtedDirect  getUsrInForInputtedDirect  accellEnt(inputtedDirect\*MoveSpeed\*deltaSinceLastFrame)  }  } |

### Planned Upgrade flowchart



### Planned PDD

## 



## Gamefeel

To add some juice to the character controller im planning on having a velocity and acceleration system for ship movement, rather then having it change directions instantly, as well as forgoing a friction system to make it feel more space/aerospace-like.

## Post Mortem

### A surprise addition

In order to keep track of our entity system and to avoid hardcoding a ship pointer into the main function, an entity store object had to be created.

This simple object with just 2 methods and one vector can be seen below, all of the entity addresses in the project are stored here allowing easy iteration at any point.

|  |
| --- |
| struct entStore {  std::vector<entClass\*> entVect = {};  void drawEntStore(sf::RenderWindow& winIn) {  for (entClass\* i : entVect) {  i->renderEnt(winIn);  }  }  void updateEntsPositions(float elapsedTimeSinceLastFrame, int xBoundMinIn, int xBoundMaxIn, int yBoundMinIn, int yBoundMaxIn) {  for (entClass\* i : entVect) {  i->updateEntPos(elapsedTimeSinceLastFrame, xBoundMinIn, xBoundMaxIn, yBoundMinIn, yBoundMaxIn);  }  }  }; |

As you can see the logic for individual methods called is kept within objects, in order to allow for custom render/movement logic for different types of object.

### A smaller addition

I also implemented my ui text element as a seprate class, for much the same reason as the entity class

|  |
| --- |
| struct madTxt {  sf::Text txt;  sf::Color fgColor;// fgColor(0, 0, 128);  sf::Color bgColor;// bgColor(0, 0, 255);  sf::Font font;  void initTxt(const char\* txtIn, int fg0, int fg1, int fg2, int bg0, int bg1, int bg2) {  // only supports comic sans rn  if (!font.loadFromFile("data/fonts/comic.ttf"))  assert(false);  //sf::Text txt(txtIn, font, 30);  txt = sf::Text(txtIn, font, 30);  txt.setPosition(10, 10);  fgColor = sf::Color(fg0, fg1, fg2);  bgColor = sf::Color(bg0, bg1, bg2);  txt.setFillColor(fgColor);  txt.setOutlineColor(bgColor);  }  }; |

### Possible improvements

There is a small bug/unintended mechanic where, due to the acceleration system, it is possible to build up velocity before reaching/while touching a boundary, leading to getting stuck shortly along a boundary sometimes.