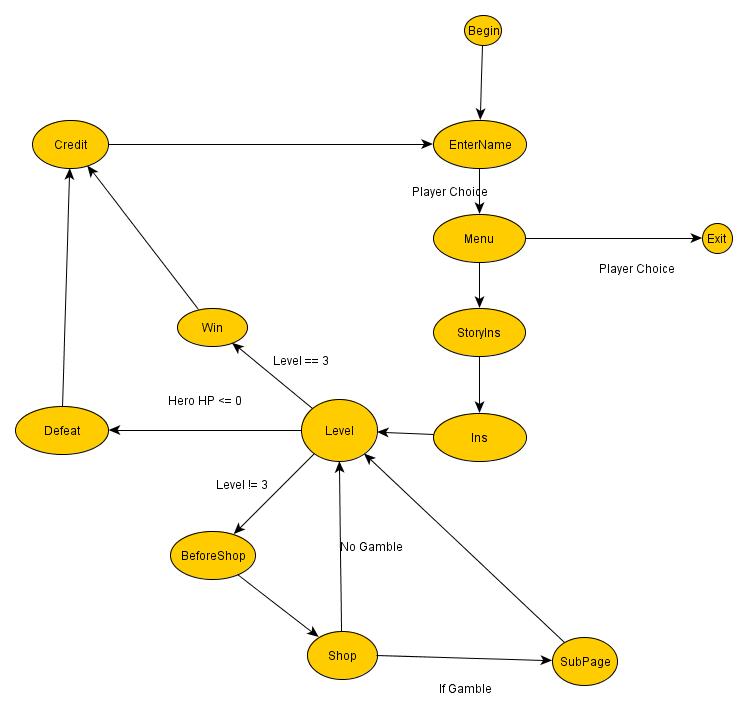
**Technical Design Document**

**The Project Framework:**

The game will be based on NeHe’s opengl codes from [www.nehe.com](http://www.nehe.com) that calls upon windows to create a window for the opengl context. The codes are split apart to exist in three parts; the Controller, the Model, and the View. The View class would be responsible for rendering different things, while the Controller would be responsible for telling the Model and the View what functions to run. The Model would do the calculations of whatever algorithms, and the updating of the objects.

To cleanly separate the game into individual parts, a State class was created. The main purpose of this state class is to determine what state the game is in, and run the corresponding code. The Controller, Model, and View all adhere to the state class. With this class, we can cleanly separate out what codes need to run where and when. Basically, the entire game is a giant state machine.

Below is our State Diagram.



Because the splitting of codes into different classes was new to most of the members, the scope of this project is relatively small. We aimed to just be able to combine our GDT assignment 1 and 2, and sprinkle some extra feature on it. Nevertheless, it proved a bit too hard.

The entire project was planned with the idea to easily run the codes of all the different classes with only a few lines. With this in mind, the planned for one entity class to cover over all the entities. By putting all these classes under a vector of type pointer Entity class, we can easily render and update these entities. The following class are located under the Entity Class

**Obstacle**

**Casino**

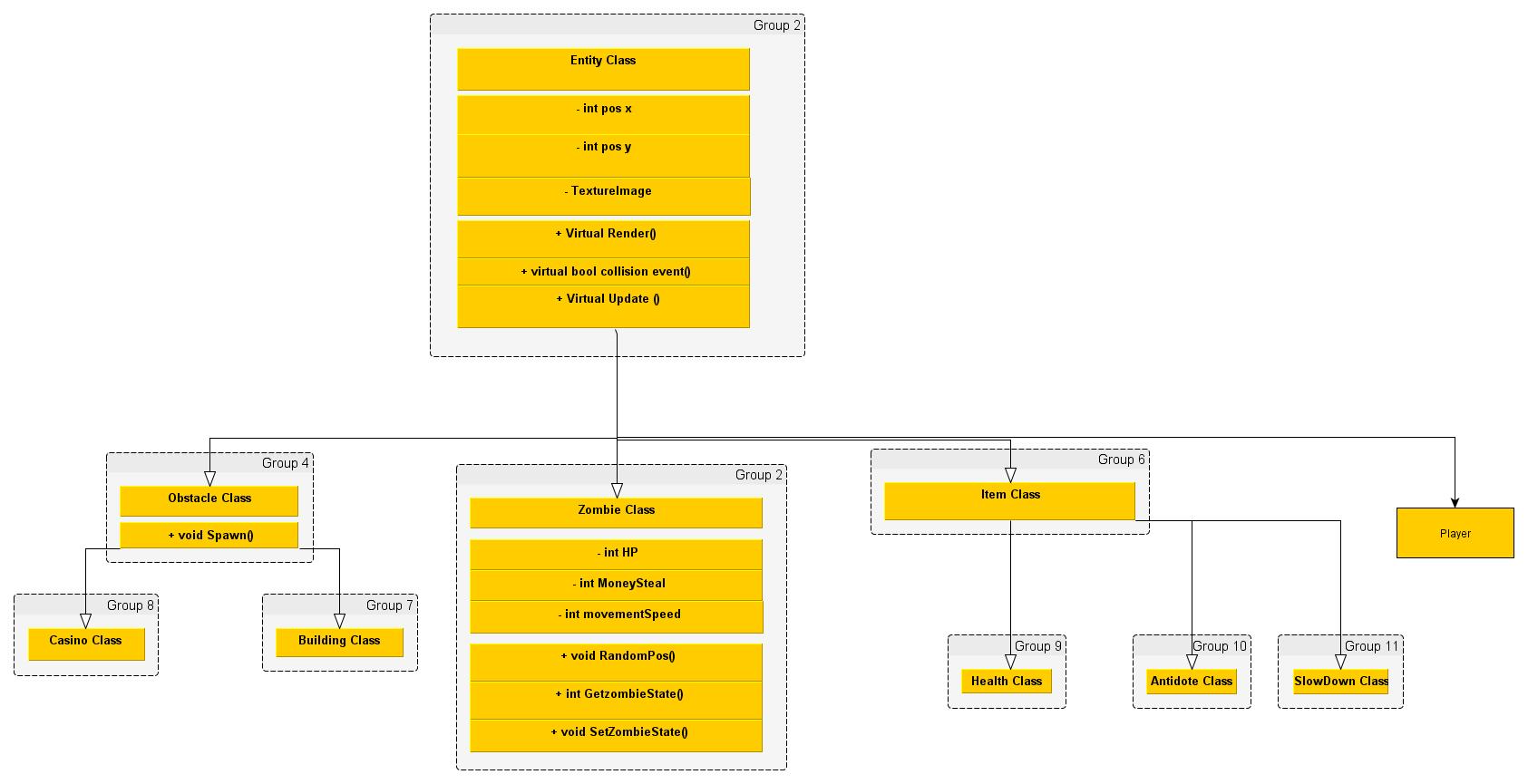
**Building**

**Zombie**

**Player**

**Items**

Below is a diagram of how the classes are arranged.



Below are what each person did on the project. This is written by themselves. There is no convention used.

**WEI JIE’S FEATURES**

- **1. Gamblers (AI)**

-**Things I’ve done**:

\* Render out the Gamblers.

+ The Gamblers uses gl-Quads to render out the zombie.

\* Update the Gamblers AI

+ It uses a Vector3 Hero Position and minus of Vector3 Gambler Position to find the relative position of the hero to the zombie. After that it is normalized to get it. After adding dt, the Zombie will not find the zombie.

- **Original** I**dea:** The gamblers are the main enemy of the game. They wondering aimlessly. Seen by them and you lose money, as they steal your money. By killing them, they can turn back to human, which is a way to save them. If the player dies at a certain level, their data will be store. So when the player replays the same level that it died, A Special gamblers with the player previous stat will spawn.

\*Normal Gamblers

+ Fast Gamblers: Movement speed is high, but attack is low.

+ Slow Gamblers: Movement Speed is low, but attack is high.

+ Normal Gamblers: Normal movement speed and attack.

- **Updated Idea**: The gamblers will now only have two gambler type. First is a normal gambler type which just find the player and attack. Second is a special gambler. It is an exact copy of the player. The other remain the same.

\* Normal Gamblers

+ It have normal movement speed and normal attack and it will find the player and attack.

\*Special Gamblers

+ This Gambler is special as it is will be almost identical to the player. The data of the player who have just died is stored and have been render it out as a special gambler.

- **2. Obstacles**

-**Things I’ve done**:

\* Render out the Obstacle.

+ Uses gl-Quads to render the casino out.

\*Update the Obstacle.

+ (Not in Use) use a function to get the amount of gambler currently in the level. It then uses an if else statement to check if the amount of zombie is less than a specific amount, it will trigger a spawn function to spawn in back till the maximum number of gambler.

- **Original** **Idea:** The two obstacle are there as a mob spawner. They will current spawn gamblers. However the building will spawn at one zombie at a time at the cost of two gamblers being killed. The Casino will start spawning gamblers, when the amount reach a certain amount, it will start spawning gamblers to the maximum amount of gamblers.

\* Building: For Every 2 gamblers kill, one gamblers will spawn

\* Casino: If the overall gamblers count reaches a certain amount, it will start to spawn gamblers to refill the overall gamblers count to the max.

- **Updated Idea:** The Obstacle now is just only a casino and will not spawn gambler. Due to some complication and time constraint, the spawning of gambler is not possible. But with enough time it will be ok.

- **3.** **Naming**

- **Original** **Idea**: Give the player a name. It is also give the Special gamblers Name.

\*Naming Convention

+ allow player to enter name

+ Entered name will appear on player In-game

**ANDY’S FEATURES**

- **Update Idea:** -No Changes-

1. Sound class

* All the sound in the game (Menu, level, Shop, Gun, Credits, etc)
* Using irrKlang sound engine to contain and play the sound
* Enumerations for all my sounds, Example:

Enum SoundType{

MENU = 0;

LEVEL,

PISTOL,

…

};

* Switch case to play the sound, Example:

Switch(Sounds)

{

case MENU:

if(Menu == NULL)

{

Menu = theSoundEngine->play2D((“Sound/Menu.wav”), true, true);

…

}

break;

}

1. LoadTGA class

* Loading of textures
* Photoshop for every texture
* Displaying of textures
* Storing textures in their specific class.
* Example: For Playerclass, storing its own texture at PlayerInfo.cpp as TextureImage heroTexture[1];

1. UI class

* UI in the game
* Every texture for every page/character.
* Using opengl knowledge
* Printw & printf
* Health bar, ammo hud

**Wan Wen’s Features**

**Entity Factory Class**

* A base class for Entities
* Creates new entities from entity class
* Example:

CEntity\* CEntityFactory::Create(Entity id)

{

CEntity\* theNewItems = NULL;

Switch(id)

{

Case HEALTH:

{

theNewItems = new CHealth;

theNewItems->ID = HEALTH;

}

Break;

Case AMMO:

{

...

}

}

}

**Entity Class**

* Get positions and set positions
* GetX
* Example :

Int CEntity::GetX( void )

{

Return pos\_x;

}

* GetY
* Example:

Int CEntity::GetY( void )

{

Return pos\_y;

}

* SetPos
* Example:

Int CEntity::SetPos( const int pos\_x, cons tint pos\_y )

{

This->pos\_x = pos\_x;

This->pos\_y = pos\_y;

}

* Virtual Functions
* Virtual void update
* Virtual void render
* Virtual void CollisionEvent

**Score (Money) Class**

* Allows player to get money in game
* AddMoney
* Example:

void CMoney::AddMoney(int addIn)

{

playerMoney = addIn + playerMoney;

}

**Items Class**

* Health class
* Creates an item that heals player’s health
* Update
* Render
  + Example:

void CHealth::render(int mapOffset\_x, int mapOffset\_y)

{

glPushMatrix();

glTranslatef(GetX() - mapOffset\_x, GetY() - mapOffset\_y,0);

glEnable( GL\_TEXTURE\_2D );

glEnable( GL\_BLEND );

glColor4f(1.0f, 1.0f, 1.0f, 1.0f);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

glColor3f(1,0.5,1);

glBegin(GL\_QUADS);

glTexCoord2f(0,0); glVertex2f(0,0);

glTexCoord2f(1,0); glVertex2f(0,tile\_size);

glTexCoord2f(1,1); glVertex2f(tile\_size,tile\_size);

glTexCoord2f(0,1); glVertex2f(tile\_size,0);

glEnd();

glDisable( GL\_BLEND );

glDisable( GL\_TEXTURE\_2D );

glPopMatrix();

}

* Ammo Class
* Creates an item that gives player extra ammo
* Update
* Render
* Speedup class
* Creates an item that allows player to speed up for a few seconds
* Update
* Render

**Player Info Class**

* Collision Event
* Speed Up
* Example:

case SLOWDOWN:

{

movementspeed = 15;

time->setActive(true,index);

..

}

**Screen textures**

* Enter Name Screen
* Menu
* Story
* Instructions
* Subpage
* Shop
* Player Gambled
* Health Pickup Item
* Speed Up

**ROLAND’S / SHUM WENG SANG FEATURES**

1. Implementing MVC base codes.

* Using NeHe’s Opengl and Mr.Toh’s base codes, I managed to create a working project using a Model,View, Controller architecture. The View is responsible for creating the window and creating the OpenGL context, and it takes care of windows’s calls. The controller would take these calls and decide what to do with it. The model holds most of the game objects, and is used mainly for math as well as updating the game.

1. Collision Class

* Implemented a collision class based on tile maps. The collision class would take two objects and create the 4 points around an object. Once it has the points, it would determine with an empirical bounding box method whether an object has collided with another.
* The object collision response code are also done by me. These codes reside in the Entity Class, with the entities themselves.
* Quad Tree – A quad tree was implemented to optimize collision because without it, the FPS drops to 5 FPS. This is simply due to the large number of entities in the game (more than 300). The quad tree is cleared every frame, and everything is inserted into it again. The quad tree would determine which node/index the object is supposed to be in. After finding the index, it would determine if it has any child nodes to put the object in, and would try to put the object in. If there are no child nodes or the object doesn’t fit in a child node, it adds the object to the parent node. Once the object is added, it determines whether the node needs to split by checking if the current number of objects exceeds the max allowed objects. Splitting will cause the node to insert any object that can fit in a child node to be added to the child node; otherwise the object will stay in the parent node. To retrieve an object, I run the entire array the holds the entities and use the retrieve function in the quad tree to get a vector of all the possible collisions. After that, I just use brute force checking with the returned vector to find collisions.
* The collision code is ran in the model function called Collision(). This function runs the quad tree and determines whether an object has collided with another. To further optimize the collision, I only run the Retrieve function on Entities that move (ie. Zombie). If collided, it will run the collision event. If an object needs to be destroyed (ie. Bullets) in the event, it will be slated for destruction with a bool. After running the collision event, another loop runs throughout the entity class that destroyed all the entities slated for destruction.

1. Mouse Class

* Created a mouse class that would take in the mouse position from Windows. I also made it so that the mouse cannot go out of the Window.

1. Gun Class (Taken over from Wei Jie)

* I’ve made a gun class that had 3 different enums to determine what type of gun it is. The gun also has bullets count and a reload function (unfortunately, not seen in the game). The gun class would spawn bullets that fire according to the power of the gun, and the movement speed of the bullet is passed in by the game. Depending on the state of the gun, it would determine for itself what type of bullets to use (3 guns 3 different types of bullets). All three guns also have different amount of bullets.

1. Vector3D Class

* Used a previously created Vector3D class for use.

1. Ortho 2D Class
   * The ortho2D class only makes the game go into 2D mode. It is only used in the draw function in the View.
2. State Class
   * The State class separated each state from another using enums. It holds a statenow, which determines the current state of the game.
3. Integration / Debugging
   * I just have to mention this. I spent more than 80% of my SP debugging other teammate’s codes and integrating it into the framework.
4. UI Clicking
   * Because there wasn’t any points for me to reference to, this is mostly done with hard code. To deal with the screen resolution, I hardcoded it into opengl Context, and than find the ratio of the screen to the opengl. This would at least make it be able to play on different computer.
5. Controller (taking in of commands)
   * Implemted a getKeys() function in the View that would retrieve the keys. I than determined the state of the game and what keys are pressed, and from there determined what to do with it.
6. Player Entity
   * The player entity would have the position of the player. It also has a Contrain Hero function that contrains the hero and moves the map according to the player’s movements.
7. Map Class
   * This class is responsible for loading the map into the game, as well as loading the entities and all the items. It holds mapOffset\_x, and mapOffset\_y that are used in the rendering of Entities to simulate a moving screen.
8. Bullet Class
   * This class uses the Vecotor3D to fire the bullet in the direction of the mouse, and update its velocity every frame so that it moves forward.

**TECHNICAL CHALLENGES AND SOLUTIONS**

One of the main problems we faced almost immediately was code rigidity. Because the entire framework depended on the use of the Entity class on a vector, it meant that codes were limited to only those on the entity class, if we were to run it on other classes. Furthermore, the entire hierarchy meant that it was quite difficult passing information around from one entity to another. To overcome this, we had to either use polymorphism or make more virtual functions. We decided to come with a mixture of both.

Another problem that came was the oversight of the game itself. To be more specific, how many entities there would be in one level. For a level to even be remotely considered a level, it contained more than 300 entities, mainly consisting of useless building classes. With this amount of entities, the FPS dropped in 5 fps and below. To optimize this, a quad tree was created.

Lastly, the members in the team did not know how the entire program flowed, even after explaining it. After teaching it to them, they slowly understood and were able to use it. This was a great hamper to the project, since Wei Jie implemented his own framework into the project. Massive time was lost in taking out the framework and fixing his feature. Because the members did not understand the framework, there were lots of bugs that appeared that they could not solve. The implementation of the classes into the Model and the View, as well as the debugging of most of the bugs could only be done by one person. This further hampered progress and gave a high load of work on Roland, since only he could integrate and do the debugging.

**CONCLUSION**

Overall, because of the misunderstood framework, we were forced to cut features as well as implement features not planned before. This greatly hampered the progress of the game. The team however did a great job at coming back with tons of things being finished at the last half of the second week and the last week.