Technical Document for

The Hills Have Towers

Version #1.2

Oct. 10th, 2012

Created by:

Pavel Szczesny

Pierre Chen

Keenan Johnston

Nicholas MacDonald

**Table of Contents**

[**History**](#vzsdjp1ovyxq)

[Version #0.0](#h50gwrq1i7sj)

[Version #1.0](#28a0dr1whqg)

[Version #1.1](#kix.z0rpwvjgfrqf)

[Version #1.2](#y6udsp9um79v)

[**Game Overview**](#y6cp7sm3fj0k)

[Game Description](#2zwk22bnf54g)

[Technology](#92vs9hl53lgg)

[Design](#ygwmfykk5q35)

[Risks](#kix.d01hnppycs4l)

[**Feature Set**](#4cd6mabpvd5h)

[Task List](#vnnkej3u2old)

[Dependency Charts](#6mdag03iseso)

[Priority Task List](#3cvh4lrs40xp)

[**Detail Technical Design**](#6bxnt4gqn5dn)

[Class Diagram](#ukmn9xvsq2gj)

[Encounters](#wwjtdb96pl8o)

[Player Encounters](#44gnhr4hqv6e)

[Tower Encounters](#7ssuxzlai1zv)

[Enemy Encounters](#6x54dncw4kqr)

[Other Encounters](#cxyokz9humzp)

[Build Tower Example Diagram](#pn8z6edk15h2)

[Event Example Diagram](#nyi1igjj1tkr)

[Tower Attack Example Diagram](#f7eb9aivke5s)

[Tower Attack Pseudocode](#bx09vm2x49xp)

[Build Tower PseudoCode](#ri1nwr5bk87s)

[High-Level Main Game Loop Pseudocode](#pr6x9vlocx5w)

[Open Path By Mining Resource Pseudocode](#hr2sxv7itisf)

[Path Finding Algorithm Pseudocode](#p6kox22c2vtq)

[Mouse Selecting Code](#phz4bpktovui)

[**Delivery Plan**](#cbmo83mxzgar)

[Deliverables](#8ea19k4e93f8)

**History**

Version #0.0: *Sept. 5th, 2012*

1. Initial draft which includes:

a. Game description

b. Technology

c. Design

d. Task List

e. Dependency Charts

f. Class Diagram

g. Encounters

h. Build Tower Example Diagram

i. Event Example Diagram

j. Tower Attack Example Diagram

k. Tower Attack Pseudocode

l.Deliverables

Version #1.0: *Sept. 8th, 2012*

1. Added the following parts to the document:

a. Build Tower Example Diagram

b. Event Example Diagram

c. Tower Attack Example Diagram

d. Encounters

2. Brokedown tasks for more detail

3. Redrew all the dependency charts

Version #1.1: *Sept. 17th, 2012*

1. Added a risks section to the document

2. Updated task list

3. Added additional code examples

Version #1.2: *Oct. 10th, 2012*

1. Added path finding algorithm pseudocode

2. Added mouse selecting code

3. Updated class diagram

4. Updated open path pseudocode and path finding

algorithm pseudocode

5. Removed task list table, picture of class diagram,

picture of dependency charts

6. Provides links for task list table, picture of class

diagram, and picture of dependency charts

7. Added priority task list

8. Updated Technology section

9. Updated Deliverables section with first three

deliverables

**Game Overview**

Game Description

The game will be a 3D tower defense game played from an isometric perspective. It will consist of levels that will feature procedurally generated terrain. Enemy waves will travel from one end of the level towards the players home base taking the shortest route available. Upon reaching the base, the enemies will capture a villager(s) and begin to take the villager back to their starting point. If an enemy carrying a villager is destroyed, the villager will slowly begin to make their way back to the base, but other enemies can recapture the villager during this period.

The player will be in control of a character which can freely move across each level. The player’s tasks will be to build towers to stop the oncoming waves of enemies. The player will also gather resources spread across the map in order to build and upgrade towers. As the player gathers resources, paths which were previously blocked by resources will open up providing enemies shorter routes between their spawn area and the players home base. In between levels, the player will also be able to purchase permanent upgrades to their character and towers.

The minimum requirements for the game will be 2GHz Intel i5 processor, Intel HD4000 graphics, and 4 GB of RAM.

Technology

All coding will be done in C++. We will also use OpenGL and programmable shaders for graphics rendering. For our graphics engine we will be using the OGRE engine. OGRE is a flexible 3D graphics engine which contains a class library that abstracts all the details of the underlying system libraries, such as OpenGL. ORGE also provides features, such as Material and Shader support, scene management, particle effects, animation support, and many other graphical rendering features. Furthermore there are a large variety of add-ons that ORGE supports, such as add-ons for lighting, creating GUIs, and a variety of physics engines.

For audio, we are going to use the FMOD Ex Programmers API. We selected FMOD because it is one of the few sound options available that has been recently updated and has two associated Ogre Sound Manager add-ons. FMOD also comes with an extensive series of demos and demonstrations that include source code. In the event that none of the Ogre Sound Manager add-ons for FMOD are functional, then the source code included in the demos will provide us with enough resources to create our own minimalist Ogre Sound Manager. An alternative for sound, OpenAL, also has an associated Ogre Sound Manager add-on, however, OpenAL hasn’t seen an updated release in over 3 years

If the need arises during development, other libraries, and add-ons will be researched and used as necessary.

Design

For our design we will be using a variation on the observer pattern through use of an EventHandler class. This class will be responsible for registering and unregistering subscribers (other objects), collecting events from other classes, and broadcasting all collected events to its subscribers so they can update their states accordingly. Instead of having classes which need to implement multiple functions to talk to other classes for multiple event types, classes will only need to be responsible for notifying the event handler that an event has occurred. Each class will then be able to handle relevant events broadcast by the event handler.

The game will be broken up into two main states, a front end state and an in game state. A Statemanger will be in charge of keeping track of these state. When in the front end state, the main menu will be active and it will also serve as the launch point to transition to the in game state. Once in the in game state, the Game class will be the main component which drives the main game loop. It will be in charge of switching between game sub-states, and its update and render functions will trigger the update and render functions for all other classes. The Game class will also be responsible for telling the event handler to broadcast all events so each class is in the correct state for updating and rendering.

All classes that will be responsible for producing a large amount of objects, such as tower classes and enemy classes, will have a corresponding manager class, such as the TowerManager and EnemyManager. These manager classes will be responsible for creating and destroying objects of their corresponding classes. In addition, it will allow us to save memory by preventing the creation of duplicate objects by instead passing a reference to the existing object and keeping a reference count for that object.

Risks

As this will be the first game to be developed by all members of the team, there will be a variety of risks associated with development. Another large risk is in regards to the technology being used for rendering graphics, and implementing physics. Once again, the tools and libraries that are being used represent a new technology that the team will need to research before beginning any implementation.

**Feature Set**

Task List

[483-Project-Task-List](https://docs.google.com/a/upei.ca/spreadsheet/ccc?key=0AsbOgN1AJmxTdDV0UV9PMk1FZHBxbEFvUkhMUXVBb3c#gid=0)



Dependency Charts

[483-Project-DependencyCharts-v2](https://docs.google.com/a/upei.ca/file/d/0B8bOgN1AJmxTUmQwOHdrRERLTVk/edit)

Priority Task List

[Priority-Task-List](https://docs.google.com/a/upei.ca/spreadsheet/ccc?key=0AsbOgN1AJmxTdDV0UV9PMk1FZHBxbEFvUkhMUXVBb3c#gid=1)

**Detail Technical Design**

Class Diagram

[483-Project-ClassDiagram-v2](https://docs.google.com/a/upei.ca/file/d/0B8bOgN1AJmxTYVZGOFdNdlRmTGM/edit)

Differences from previous version:

1. Change all "getStatement", "setStatement" to "handleEvent" function.

2. Update functions in EventHandler class.

3. Add "loop" function into Game class.

4. Add Event class.

5. Add SelectableManager, Selectable, and RayQuery class for mouse selecting algorithm.

6. Remove SupportTower and SupportTowerManager class.

7. Add StateManager class.

8 SupportEnemy class now only have "heal" function.

9. Add SlowTower class, which extends BaseTower class

10. Add “size” function to TowerManager class

11. Add “setTarget” function to BaseTower class

12. Rename Subject class to Recipient

Encounters

**Player Encounters**

**Player runs into enemy**

● Player gets pushed back and knocked down and remains

immobile for a couple seconds.

**Player runs into terrain (mountains, water, etc.)**

● Players progress is halted as terrain is blocking the path

● If the terrain tile is suitable, the player can build a tower in the tile

**Player runs into tower**

● Players progress is not halted as player can move freely through

their own towers

● Player has the ability to sell the tower

● Player has the ability to upgrade the tower provided the player

has enough resources

**Player moving over free tiles**

● Player has the ability to build a tower provided the player has

enough resources

● Player can continue moving through tile

**Player runs into resources**

● Players progress is halted as resource is blocking the path

● Player has the ability to harvest resource

**Player tries to build a tower**

● A check is done to determine if currently tiles is valid for tower

building

● A check is done to determine what towers can be built based on

the players current resources

● A menu displaying towers that can be currently built is shown.

Towers which cannot be built are also displayed, but their selection is disabled and required resources to build the disabled towers are displayed

**Player initiates a tower build**

● Resources are removed from the player as per the cost of the

tower

● Player is free to move away as the tower will auto build to

completion

**Player selects a tower**

● The towers attack radius gets displayed

● A menu pops up and displays the following three options:

Salvage the tower and how many resources the player will

receive for salvaging, tower information, and upgrade tower and

the amount of resources required to upgrade the tower

● Player can click on a enemy to make the tower focus its attack on

that enemy

**Player initiates a tower upgrade**

● A check is done to see if the player has enough resources to

upgrade the tower

● If the check passes, resources are removed from the player as

per the cost of the upgrade

● Player is free to move away as the tower will auto upgrade to

completion

**Player initiates a tower salvage**

● The tower gets destroyed

● Resources are added to the players total resources

**Player views tower information**

● A tool-tip style pop-up appears displayed the tower attack power

and rate of fire

**Player harvests resources**

● Resources of the type being harvested get added to the players

total count

● Player will automatically continue harvesting the current resource

until the resource is depleted or is moved away from the resource

**Tower Encounters**

**Tower gets built**

● Build animation plays and upon completion, tower enters idle

state

**Tower sitting idle**

● Tower continually checks to see if an enemy has entered its

attack radius

**Enemy enters towers attack radius**

● Tower switches from idle state to its attack state and begins to

auto fire

● Towers will prioritizes attack on a first to enter the attack radius

basis

**Tower kills an enemy**

● If there are more enemies in the tower’s attack radius, tower

continues to attack prioritizing enemies on a first to enter the

attack radius basis

● If there are no enemies left in the attack radius, tower switches to

an idle state

**Enemy leaves towers attack radius**

● If there are more enemies in the tower’s attack radius, tower

continues to attack prioritizing enemies on a first to enter the

attack radius basis

● If there are no enemies left in the attack radius, tower switches to

an idle state

**Tower gets upgraded**

● If tower is in an attack state, tower switches to an idle state and

begins its upgrade animation

● Once upgrade animation is done, tower returns to an idle state

**Tower gets sold**

● If tower is in an attack state, tower switches to an idle state and

begins its salvage animation

**Enemy Encounters**

**Enemy enters level**

● Enemy calculates shortest path to player base

● Enemy moves toward player base following the shortest path

**Enemy moving**

● If enemy receives a resource depleted event, enemy recalculates

the shortest path to the player base/spawn point

● Enemy continues moving toward player base/spawn point

following the shortest path

**Enemy reaches player base**

● Enemy grabs villager and begins returning to its spawn location

following the shortest path

● If no villager is present at the player base, enemy begins returning

to its spawn location following the shortest path

**Enemy runs into player**

● Enemy knocks player aside and continues moving toward the

player base/spawn point following the shortest path

**Enemy runs into villager**

● If enemy is carrying its max amount of villagers, enemy ignores

villager and continues returning to spawn point following the

shortest path

● If enemy can carry villager, enemy grabs villager and then begins

or continues returning to spawn point following the shortest path

**Enemy gets attacked by tower**

● If attack is a damaging attack, the amount of the attack damage is

subtracted from the enemies remaining health

● If attack damage is more than enemies remaining health, enemy

enters death state

● If attack is a slowing attack, enemies movement speed is reduced

temporarily

**Enemy enters death state**

● If enemy is carrying villager(s), enemy drop villager(s)

● Enemy death animation plays

**Other Encounters**

**Villager gets dropped by enemy**

● Villager begins walking back to player base following the shortest

path

**Villager walking back to player base**

● If villager receives a resource depleted event, villager recalculates

the shortest path to the player base

● Villager continues moving toward player base following the

shortest path

**Resource depleted**

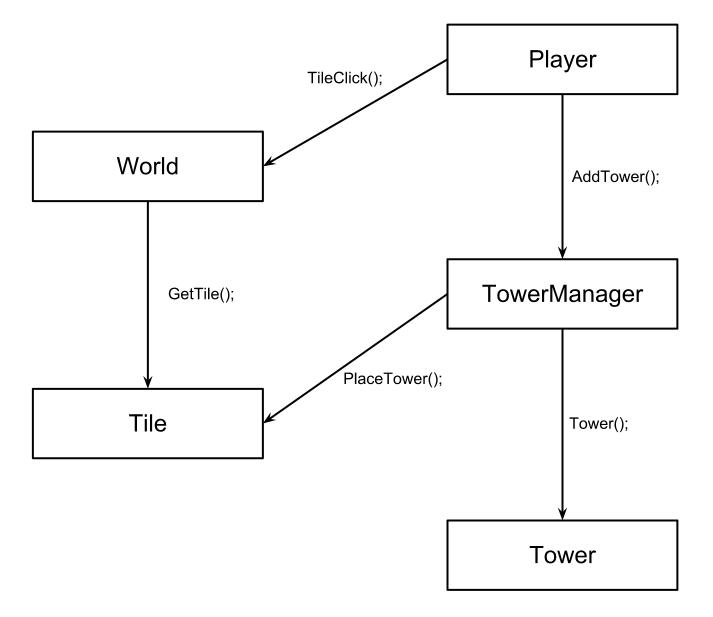
● Resource depletion animation plays

● Tile occupied by resource becomes empty but remains in a non

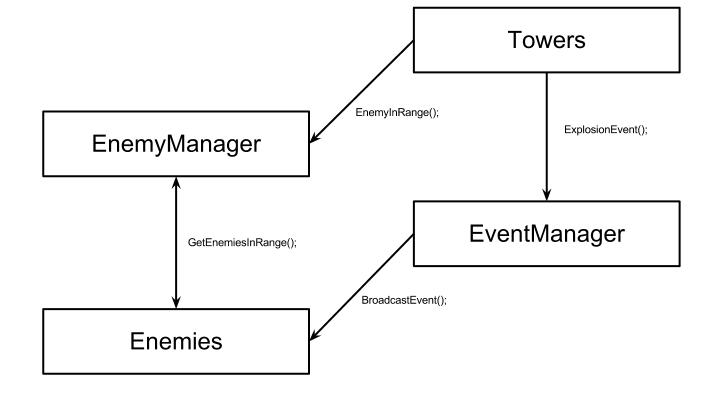
buildable state

● Resource depletion event gets sent to event handler

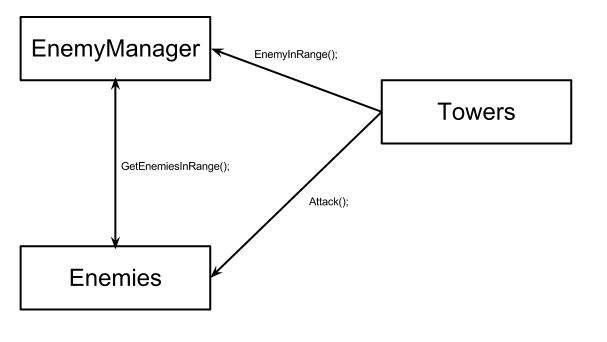
Build Tower Example Diagram



Event Example Diagram



Tower Attack Example Diagram



Tower Attack Pseudocode

// Pseudo code for tower attack

ArrowTower::EnemyInRange()

{

if (EnemyManager::EnemiesInRange(m\_fRange))

{

m\_pEnemies = EnemyManager::GetEnemiesInRange(m\_fRange);

m\_eState = AttackState;

}

}

ArrowTower::Attack()

{

if (m\_iCurrentTarget < pEnemies->size())

{

if(pEnemies[m\_iCurrentTarget] != dead && pEnemies[m\_iCurrentTarget]->InRange(m\_fRange))

{

pEnemies[i]->Attacked(m\_iAttackPower);

}

else

{

m\_iCurrentTarget++;

}

}

else

{

m\_eState = IdleState;

m\_iCurrentTarget = 0;

}

}

ArrowTower::Update()

{

if (m\_eState == AttackState)

{

Attack();

}

else

{

EnemyInRange();

}

// other update code

}

// Pseudo code for an Explode Event

BombTower::EnemyInRange()

{

if (EnemyManager::EnemiesInRange(m\_fRange))

{

m\_pEnemies = EnemyManager::GetEnemiesInRange(m\_fRange);

m\_eState = AttackState;

}

}

BombTower::Attack()

{

if (m\_iCurrentTarget < pEnemies->size())

{

if(pEnemies[m\_iCurrentTarget] != dead && pEnemies[m\_iCurrentTarget]->InRange(m\_fRange))

{

EventHandler::AddEvent(new Event(ExplodeEvent));

}

else

{

m\_iCurrentTarget++;

}

}

else

{

m\_eState = IdleState;

m\_iCurrentTarget = 0;

}

}

BombTower::Update()

{

if (m\_eState == AttackState)

{

Attack();

}

else

{

EnemyInRange();

}

// other update code

}

EventHandler::AddEvent(pEvent)

{

// Code for adding events

}

EventHandler::Update()

{

BroadcastEvents();

}

Enemy::HandleEvent(pEvent)

{

// code for handling event

}

FireEnemy::HandleEvent(pEvent)

{

// code for handling event different from how a normal enemy handles an explode event

}

Build Tower PseudoCode

// Pseudo code for building a tower

Player::TileClick(p\_fPos)

{

Tile\* pTile = World->GetTile(p\_fPos);

if (!pTile->HasTower() && pTile->IsValid())

{

// code for selecting tower to build

TowerManager::AddTower(p\_fPos);

}

else if (pTilee->HasTower())

{

// code for upgrading towers

}

else

{

// code for handling clicking on invalid tiles

}

}

TowerManager::AddTower(p\_fPos)

{

Tower\* pTower = new Tower();

Tile\* pTile = World::GetTile(pfPos);

pTile->PlaceTower(pTower);

towers.push\_back(pTower);

}

High-Level Main Game Loop Pseudocode

// The rendering time intervals, need to be decided

const int FRAMEPERSECOND = FramePerSecond

// In Game class, need to include <ctime>

Game::Init()

{

// Initialization code

Loop();

}

Game::Loop()

{

while(!end\_game\_requested)

{

// Pause game only after one rendering time interval passed

if(!Game::pauseState)

{

Game::Update();

Game::Render();

}

}

}

Game::Update()

{

// Calculate when to stop updating and to start rendering

clock\_t endOfUpdate = clock() + static\_cast<clock\_t>(FRAMEPERSECOND);

// Keep updating until it's time to render window

while( clock() < endOfUdate)

{

// Call update method of each entity

}

}

Game::Render()

{

// Tell Ogre to render window

}

// Game class also need to extends OIS class in order to take care the input from player.

Open Path By Mining Resource Pseudocode

//In Player class

Player::TileClick(p\_fPos)

{

Tile\* pTile = World->GetTile(p\_fPos);

if (pTile->HasResource())

{

m\_eState = HarvestState;

m\_pTileToHarvest = pTile;

}

// other conditional statements

}

Player::Harvest()

{

Resource\* pHavesting = m\_pTileToHarvest->GetResource();

if (pHavesting->VeinSize() > 0)

{

// Code for harvesting resource

}

else {

m\_pTileToHarvest->ClearResource();

m\_eState = IdleState;

if (m\_pTileToHarvest->HasPath())

{

EventHandler::AddEvent(new Event(OpenPath), params);

}

}

}

Player::Update()

{

if (m\_eState == HarvestState)

{

Harvest();

}

// other update code

}

// In EventHandler class

EventHandler::Update()

{

BroadcastEvent();

}

// In Enemy class

Enemy::HandleEvent(pEvent)

{

// code for handling events

}

// In Villager class

Villager::HandleEvent(pEvent)

{

// code for handling events

}

// Assume World class contains XML file which stores the information about the main path and all the other branch paths.

// For each branch path, it also contains the location(s) where it is blocked, and if the branch currently blocked or not

//In World class

World::HandleEvent(pEvent)

{

// code for handling a new path opening

}

Path Finding Algorithm Pseudocode

[Link: Pathfinding mockup](https://docs.google.com/a/upei.ca/document/d/1JtkQnRTe_gae0szF-m_zvT9hfOxq1UxLWdZmjdMLUM4/edit)

[Link: Pathfinding code and result](https://docs.google.com/a/upei.ca/document/d/1PcUMwDh5y_f0LqFX4TjDwHQB_QspFJXhfilrglWXXOs/edit)

// In world class

World::handleEvent()

{

// If receive OpenPath event

struct tileCounter

{

Tile\* tile;

int steps;

}

//List of tiles that we have already visited and left behind

list<tileCounter> visitedList;

//List of all adjacent tiles that have yet to be explored

list<tileCounter> currentList;

//Our search begins at the end tile and we search for the start

currentList.push\_back(tileCounter(&VillageTile, 0));

//We declare that we haven't found the start yet and we haven't progressed

//to any tiles

bool startFound = false;

int tilesTraveled = 0;

while(!startFound)

{

vector<tileCounter>::iterator it;

//for every tile we're currently standing on with unexplored adjacent

//tiles

for(it = currentList.begin(); it < currentList.end(); it++ &&

!startFound)

{

tileCounter currentTile = \*it;

//If we found the starting tile

if(currentTile.tile == mStartingTile)

{

//We record how many steps it took us to get to this tile

//and that we found it

tilesTraveled = currentTile.steps;

startFound = true;

}

//Check all adjacent tiles (not counting diagonal ones or tiles

//that we've already been to (exist in visitedList))

//If we haven't found the starting tile and we can move to an

//unvisited adjacent tile, then we add it to the list of tiles

//we're standing on

if(adjacentTile == PATH\_TILE && !startFound)

currentList.push\_back(tileCounter(adjacentTile,

currentTile.steps++);

//We remove the current tile from our list of current tiles and

//add it to our list of previously visited tiles

visitedList.push\_back(currentTile);

currentList.remove(currentTile);

}

}

queue<Tile> shortestPath;

while(tilesTraveled != 0)

{

//Starting at the starting tile, find the adjacent tile with a step

//count of one less then the current tile

//Move to that tile and add the previous tile to the shortestPath queue

//Decrement the tilesTraveled (once it reaches 0, we'll have reached

//the exit which is where we started our A\* algorithm

}

//We send the finished shortest path, a queue, to the enemy manager so

//that it can properly inform all enemies about the new path

EventHandler::AddEvent(new Event(NewPathEvent));

}

//The enemy’s handle this by checking if they’re currently on any of the tiles in the queue. When they find the tile that they’re on, they pop tiles off the queue until they match the one they’re on. They then progress down the rest of the newly generated path.

Mouse Selecting Code

[Link: Mouse selecting code](https://docs.google.com/a/upei.ca/file/d/0B8bOgN1AJmxTeFZSZVhSZDQzY1k/edit)

// Game has to extend OIS::MouseListener

// In Game class

Game::mouseMoved(const OIS::MouseEvent& arg)

{

// Let CEGUI know that the mouse moved

CEGUI::System::getSingleton().injectMouseMove(arg.state.X.rel, arg.state.Y.rel);

// Get the mouse from CEGUI

CEGUI::MouseCursor\* pMouse = CEGUI::MouseCursor::getSingletonPtr();

// Create a ray depends on Camera. The parameters are normalised x and y coordinate on screen(viewport)

Ogre::Ray mMouseRay = m\_pCamera->getCameraToViewportRay(pMouse->getPosition().d\_x / float(arg.state.width), mouse->getPosition().d\_y / float(arg.state.height));

// Setup RayQuery class by given Ray

m\_pRayQuery->SetRay(mMouseRay);

// Fire RayQuery to find the Selectable object

Selectable\* pSelectableObj = m\_pRayQuery->execute();

// Do things we want to do for the object

}

Game::mousePressed(const OIS::MouseEvent& arg, OIS::MouseButtonID id)

{

// Get the mouse from CEGUI

CEGUI::MouseCursor\* mouse = CEGUI::MouseCursor::getSingletonPtr();

// Create a ray depends on Camera. The parameters are normalised x and y coordinate on screen(viewport)

Ogre::Ray mouseRay = m\_pCamera->getCameraToViewportRay(mouse->getPosition().d\_x / float(arg.state.width), mouse->getPosition().d\_y / float(arg.state.height));

// Create a ray depends on Camera. The parameters are normalised x and y coordinate on screen(viewport)

Ogre::Ray mMouseRay = m\_pCamera->getCameraToViewportRay(pMouse->getPosition().d\_x / float(arg.state.width), mouse->getPosition().d\_y / float(arg.state.height));

// Setup RayQuery class by given Ray

m\_pRayQuery->SetRay(mMouseRay);

// Fire RayQuery to find the Selectable object

Selectable\* pSelectableObj = m\_pRayQuery->execute();

// Do things we want to do for the object

}

Game::mouseRelease(const OIS::MouseEvent& arg, OIS::MouseButtonID id)

{

// Get the mouse from CEGUI

CEGUI::MouseCursor\* mouse = CEGUI::MouseCursor::getSingletonPtr();

// Create a ray depends on Camera. The parameters are normalised x and y coordinate on screen(viewport)

Ogre::Ray mouseRay = m\_pCamera->getCameraToViewportRay(mouse->getPosition().d\_x / float(arg.state.width), mouse->getPosition().d\_y / float(arg.state.height));

// Create a ray depends on Camera. The parameters are normalised x and y coordinate on screen(viewport)

Ogre::Ray mMouseRay = m\_pCamera->getCameraToViewportRay(pMouse->getPosition().d\_x / float(arg.state.width), mouse->getPosition().d\_y / float(arg.state.height));

// Setup RayQuery class by given Ray

m\_pRayQuery->SetRay(mMouseRay);

// Fire RayQuery to find the Selectable object

Selectable\* pSelectableObj = m\_pRayQuery->execute();

// Do things we want to do for the object

}

// In RayQuery class

Selectable\* RayQuery::execute(){

pClosestSelectableObj = 0; // Reset the pointer of closest object

mClosestSelectableObjDistance = 0; // Reset the distance between closest object and camera

// For each selectable object in the list

for(int i = 0; i < SelectableManager::size(); i++){

// If the selectable object is selectable

if(SelectableManager::getSelectableObj(i)->isSelectable()){

// Create a bounding box by giving the minima point ((x, y, z) = (left, botton, in)) and maxima point ((x, y, z) = (right, top, out)) of the selectable object

Ogre::AxisAlignedBox boundingBox = Ogre::AxisAlignedBox(SelectableManager::getSelectableObj(i)->getMinPoint, SelectableManager::getSelectableObj(i)->getMaxPoint);

std::pair<bool, Ogre::Real> intersection = mRay.intersects(boundingBox);

// If intersect, compare the distance of current object with the distance of current closest object.

if(intersection.first){

// If current object is closer than current closest object, change current closest object to current object

if(mClosestSelectableObjDistance > intersect.second){

pClosestSelectableObj = SelectableManager::getSelectableObj(i);

mClosestSelectableObjDistance = intersect.second;

}

}

}

}

return pClosestSelectableObj;

}

// In SelectableManager

Selectable\* SelectableManager::getSelectableObj(int index){

return m\_pSelectableList(index);

}

int SelectableManager::size(){

return mSize;

}

// In Selectable

Ogre::Vector3 Selectable::getMinPoint(){

return mMinPoint;

}

Ogre::Vector3 Selectable::getMaxPoint(){

return mMaxPoint;

}

**Delivery Plan**

Deliverables

**Sprint 1 -** Begin terrain generation and base level (boxes on a plane) game implementation. Implement event handler and events.

**Sprint 2 -** Continue with terrain generation and base level game implementation. Begin creation of in game HUD and menus.

**Sprint 3 -** Complete terrain generation and base level game implementation. Continue creating in game HUD and menus.

**Sprint 4 -**

**Sprint 5 -**

**Sprint 6 -**

**Sprint 7 -**

**Sprint 8 -**