

ACID Technical Design Document

All work Copyright © 2018 by Ian Christopher Apino, David Tea, Alvin Tang and Callum Drennan

Written by Ian Christopher Apino, David Tea, Alvin Tang and Callum Drennan

Version: 1.0

Technical Goals

- User-Friendly interfaces
- Responsive physics-based gameplay
- Robust randomized level generation
- AI that is actually intelligent
- Pretty particle systems
- Comprehensive gore system

Libraries

- GLEW <http://glew.sourceforge.net/>
- GLM (OpenGL Mathematics) <https://glm.g-truc.net/0.9.9/index.html>
- picoPNG <https://lodev.org/lodepng/>
- SDL <https://www.libsdl.org/>
- OpenGL
- FMOD <https://www.fmod.com/>
- Box2D <https://box2d.org/>

SVN Standards

- Do not commit when there are present syntax or runtime errors.
- Always update from the repository when working on the project.
- Commits must have clear and meaningful messages.

Coding Standards

- Null pointers defined as 0
- `#ifndef` and `#define` with `#pragma once`

Naming Conventions

- camelCasing for variable names.
- PascalCasing for method/function names.
- `m_` prefix for member fields
- `m_p` prefix for pointer member fields
- Method/function names
- Enums should be all caps and separated with an underscore (`MOVE_LEFT`)
- Booleans must be prefixed with a lowercase `b.` (`blsDying`)

Commenting

- Write relevant comments
- Write useful comments
- Comment personal code regularly

Data Structures

- Sprite Batches
- Object Pools

OpenGL

Installing GLEW

GLEW Library:

The libraries are to be downloaded and placed in a ExternalLibraries folder on the same directory as the solution file.

The project can now be configured to include the include directories of those libraries.

If the libraries aren't there to link our project to them via the linker property settings, then it can be built manually.

Physics

The game will be using the Box2D library to handle movement, collisions and environmental interactions. The library files for Box2D can be found in the SVN shared folder.

The Physics World

Because ACID is a top down game, gravity should not affect the game objects. The gravity scale can be set to 0.0f when initialising the world.

Movement

There will be two movement states; horizontal and vertical. The player will store a current movement state for both, to allow diagonal movement.

```
enum class VerticalMovementStates
{
    NONE, UP, DOWN
}
```

```
enum class HorizontalMovementStates
{
    NONE, LEFT, RIGHT
}
```

Creating a Body

To create an object with physics properties in it, we must first create a body. A body is just an invisible object that has no shape. It defines the position and angle of the object. A shape can be created, and attached to the body with a fixture. The shape is well... the shape. The fixture is the properties of the shape, such as friction, bounciness and density.

```
// Creating a simple box
b2BodyDef bodyDef; // Defines body
bodyDef.type = b2_dynamicBody // Sets the body type to be dynamic

// m_body - b2Body declared in header
// world - b2World passed in from game
m_body = world->CreateBody(&bodyDef); // Creates the body

b2PolygonShape shape; // Defines the shape, in this case, a polygon
// or b2CircleShape for circle shape
shape.SetAsBox(width, height); // Automatically sets the shape with 4
points. Width and height is center-out.

b2FixtureDef fixtureDef; // Define fixture
fixtureDef.shape = &shape; // Attaches shape to fixture
fixtureDef.density = 1.0f; // Sets the density
fixtureDef.friction = 1.0f; // Sets the friction

m_body->CreateFixture(&fixtureDef); // Attaches fixture to the body
```

```
// Creating a wall
b2BodyDef bodyDef;
bodyDef.type = b2_staticBody; // Static because walls shouldn't move
bodyDef.position.Set(x, y); // Can also set the position on initialisation
of body
m_body = world->CreateBody(&bodyDef);

b2PolygonShape shape;
shape.SetAsBox(width, height);
m_body->CreateFixture(&shape, 0.0f); // Create fixture with default 0.0f
density
```

Collision Detection

By default, all physics object can collide with each other. We can define what objects and collide with each other and which ones cannot by using category and mask bits. Category bits is what defines the type of object, and mask bits are the types of objects it can collide with. We can only have a maximum of 16 categories, because 256 bits.

Defined below will be the game's current category bits and part of the PhysicsEntity class.

```
enum CollisionBits
{
    C_PLAYER,
    C_OBJECTS,
    C_ENEMIES,
    C_WEAPONS,
    C_TRAPS,
    C_WALLS
}
```

The category and mask bits can be defined when creating the fixture.

```
// Category and Mask bits for the Player
// Can be defined in the initialiser of the Player class
m_categoryBits = C_PLAYER;
m_maskBits = C_WALL | C_WEAPONS | C_ENEMIES | C_OBJECTS | C_TRAPS

// Assigning the category and mask bits for the Character class
// While creating the fixture:
fixtureDef.filter.categoryBits = m_categoryBits;
fixtureDef.filter.maskBits = m_maskBits;
```

Collision Callbacks

When certain objects collide with each other, such as a weapon and an enemy, we want something to happen; like taking damage. We can implement the b2ContactListener interface and then add it to our physics world.

There are four functions that can be implemented:

```
// Called as both objects collide with each other
void BeginContact(b2Contact* contact);
// Called both objects stop colliding with each other
void EndContact(b2Contact* contact);
// Called after the collision
void PostSolve(b2Contact* contact, const b2ContactImpulse* impulse);
// Called before the collision
void PreSolve(b2Contact* contact, const b2Manifold* oldManifold);
```

Vertex and Fragment Shaders

There will be a GLSL class that handles the linking and compilation of the vertex and fragment shaders. The vertex shader will take in the vertex position, color and UV from the game class these are then sent to the fragment shader.

Vertex Shader:

```
#version 130

in vec2 vertexPosition;
in vec4 vertexColor;
in vec2 vertexUV;

out vec2 fragmentPosition;
out vec4 fragmentColor;
out vec2 fragmentUV;

uniform mat4 P;

void main()
{
    //Set the x,y position on the screen
    gl_Position.xy = (P * vec4(vertexPosition, 0.0, 1.0)).xy;
    //the z position is zero since we are in 2D
    gl_Position.z = 0.0;

    gl_Position.w = 1.0;

    fragmentPosition = vertexPosition;

    fragmentColor = vertexColor;

    fragmentUV = vec2(vertexUV.x, 1.0 - vertexUV.y);
}
```

Fragment Shader:

```
#version 130

in vec2 fragmentPosition;
in vec4 fragmentColor;
in vec2 fragmentUV;

out vec4 color;

uniform sampler2D mySampler;
```

```
void main()
{
    vec4 textureColor = texture(mySampler, fragmentUV);

    color = fragmentColor * textureColor;
}
```

Image Loading and Sprites

Resource Manager and Texture Cache:

A resource manager will be in charge of all the textures used in the game. The resource manager will have a texture cache which loads in a PNG image using picoPNG(third party open source image encoding and decoding class). A texture map will be used to ensure a texture is not loaded more than once.

```
GLTexture TextureCache::getTexture(std::string texturePath)
{
    auto iter = _textureMap.find(texturePath);

    //Check if texture is in the map
    if (iter == _textureMap.end())
    {
        //Load Texture
        GLTexture newTexture = ImageLoader::loadPNG(texturePath);

        //Insert texture into map
        _textureMap.insert(make_pair(texturePath, newTexture));

        return newTexture;
    }

    return iter->second;
}
```

```
GLTexture ImageLoader::loadPNG(std::string filePath)
{
    GLTexture texture = {};

    std::vector<unsigned char> in;
    std::vector<unsigned char> out;

    unsigned long width, height;

    if (IOManager::readFileToBuffer(filePath, in) == false)
    {

```



```

        fatalError("Failed to load PNG to buffer");
    }

    int errorCode = decodePNG(out, width, height, &(in[0]), in.size());

    if (errorCode != 0)
    {
        fatalError("decodePNG failed with error: "
+std::to_string(errorCode));
    }

    glGenTextures(1, &(texture.id));

    glBindTexture(GL_TEXTURE_2D, texture.id);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, &(out[0]));

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    glGenerateMipmap(GL_TEXTURE_2D);

    glBindTexture(GL_TEXTURE_2D, 0);

    return texture;
}

```

The sprite will take in x, y, width, height and texture parameters. A sprite will use a vertex struct which contains the position(x, y), the color (r,g,b) and UV (u,v). This vertex will be used as the buffer data.

```

void Sprite::init(float x, float y, float width, float height, std::string
texturePath)
{
    _x = x;
    _y = y;
    _width = width;
    _height = height;
    _texture = ResourceManager::getTexture(texturePath);

    if (_vboID == 0)
    {
        glGenBuffers(1, &_vboID);
    }

    Vertex vertexData[6];

```

```
vertexData[0].setPosition(x + width, y + height);
vertexData[0].setUV(1.0f, 1.0f);

vertexData[1].setPosition(x, y + height);
vertexData[1].setUV(0.0f, 1.0f);

vertexData[2].setPosition(x, y);
vertexData[2].setUV(0.0f, 0.0f);

vertexData[3].setPosition(x, y);
vertexData[3].setUV(0.0f, 0.0f);

vertexData[4].setPosition(x + width, y);
vertexData[4].setUV(1.0f, 0.0f);

vertexData[5].setPosition(x + width, y + height);
vertexData[5].setUV(1.0f, 1.0f);

for (int i = 0; i < 6; i++)
{
    vertexData[i].setColor(255, 0, 255, 255);
}

vertexData[1].setColor(0,0, 255, 255);

vertexData[4].setColor(0, 255, 0, 255);

glBindBuffer(GL_ARRAY_BUFFER, _vboID);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertexData), vertexData,
GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

The sprite draw method will bind a texture then bind the vbo to the buffer. The vertex data will then be used to draw sprite.

```
void Sprite::draw()
{
    glBindTexture(GL_TEXTURE_2D, _texture.id);

    glBindBuffer(GL_ARRAY_BUFFER, _vboID);

    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);
    glEnableVertexAttribArray(2);

    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, position));
    glVertexAttribPointer(1, 4, GL_UNSIGNED_BYTE, GL_TRUE,
sizeof(Vertex), (void*)offsetof(Vertex, color));
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, uv));

    glDrawArrays(GL_TRIANGLES, 0, 6);

    glDisableVertexAttribArray(0);
    glDisableVertexAttribArray(1);
    glDisableVertexAttribArray(2);

    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

Lighting

Lighting vertex and fragment shaders will be used to display a simple 2D light. A light sprite will hold the position, size and color of the light source.

Fragment Shader:

```
#version 130

in vec2 fragmentPosition;
in vec4 fragmentColor;
in vec2 fragmentUV;

out vec4 color;

void main() {
```

```
float distance = length(fragmentUV);
color = vec4(fragmentColor.rgb, fragmentColor.a * (pow(0.01, distance)
- 0.01));
}
```

Vertex Shader:

```
#version 130
//The vertex shader operates on each vertex

//input data from the VBO. Each vertex is 2 floats
in vec2 vertexPosition;
in vec4 vertexColor;
in vec2 vertexUV;

out vec2 fragmentPosition;
out vec4 fragmentColor;
out vec2 fragmentUV;

uniform mat4 P;

void main() {
    //Set the x,y position on the screen
    gl_Position.xy = (P * vec4(vertexPosition, 0.0, 1.0)).xy;
    //the z position is zero since we are in 2D
    gl_Position.z = 0.0;

    //Indicate that the coordinates are normalized
    gl_Position.w = 1.0;

    fragmentPosition = vertexPosition;

    fragmentColor = vertexColor;

    fragmentUV = vertexUV;
}
```

Camera

The camera will use a camera matrix and an orthographic matrix with the help of GLM. Camera movement is handled by an update method that translates and scales camera matrix based on the user input.

```

void Camera::init(int screenWidth, int screenHeight)
{
    _screenWidth = screenWidth;
    _screenHeight = screenHeight;

    //Set Camera Origin Points
    setOrigin(glm::vec2(_screenWidth/2, _screenHeight/2));
    orthoMatrix = glm::ortho(0.0f, (float)_screenWidth, 0.0f,
(float)_screenHeight);
    glm::vec3 translate(origin, 0.0f);
    cameraMatrix = glm::translate(orthoMatrix, translate);
    glm::vec3 scale(scale, scale, 0.0f);
    cameraMatrix = glm::scale(glm::mat4(1.0f), scale) * cameraMatrix;
    matrixUpdate = false;
}

```

```

void Camera::update()
{
    if (matrixUpdate)
    {
        glm::vec3 translate(-position.x + _screenWidth/2, -position.y +
_screenHeight/2 , 0.0f);
        cameraMatrix = glm::translate(orthoMatrix, translate);

        //Camera Scale
        glm::vec3 scale(scale, scale, 0.0f);
        cameraMatrix = glm::scale(glm::mat4(1.0f), scale) *
cameraMatrix;

        matrixUpdate = false;
    }
}

```

```

case SDL_KEYDOWN:
    switch (e.key.keysym.sym)
    {
        case SDLK_w:
            mainCamera.setPosition(mainCamera.getPosition() + glm::vec2(0.0,
CAMERASPEED));
            break;
        case SDLK_s:
            mainCamera.setPosition(mainCamera.getPosition() + glm::vec2(0.0, -
CAMERASPEED));
            break;
        case SDLK_d:
            mainCamera.setPosition(mainCamera.getPosition() +

```

```

glm::vec2(CAMERASPEED, 0.0));
    break;
    case SDLK_a:
        mainCamera.setPosition(mainCamera.getPosition() + glm::vec2(-
CAMERASPEED, 0.0 ));
    break;
    case SDLK_q:
        mainCamera.setScale(mainCamera.getScale() + SCALESPEED);
    break;
    case SDLK_e:
        mainCamera.setScale(mainCamera.getScale() - SCALESPEED);
    break;
}

```

Sprite Batching

To improve the game's performance sprite batching will be used to pass in the sprites through a render batch to the GPU instead of passing each sprite one by one. A member object represents a single sprite.

```

class Member
{
    public:
        Member()
        {
        };
        Member(const glm::vec4& destRect, const glm::vec4& uvRect,
GLuint Texture, float Depth, const ColorRGB& color);
        Member(const glm::vec4& destRect, const glm::vec4& uvRect,
GLuint Texture, float Depth, const ColorRGB& color, float angle);

        GLuint texture;
        float depth;

        Vertex topLeft;
        Vertex bottomLeft;
        Vertex topRight;
        Vertex bottomRight;
    private:
        glm::vec2 rotatePoint(const glm::vec2& pos, float angle);
};

```

The `emplace_back` method of `std::vector` will be used to instantiate a new render batch and sprite at the end of the vector with member pointer variables as a parameter (used for both vertices and texture).

```
void SpriteBatch::createRenderBatches()
{
    std::vector<Vertex> vertices;
    vertices.resize(m_memberPointers.size() * 6);

    if (m_memberPointers.empty())
    {
        return;
    }

    int offset = 0;
    int cVert = 0;

    m_renderBatches.emplace_back(offset, 6, m_memberPointers[0]->texture);
    vertices[cVert++] = m_memberPointers[0]->topLeft;
    vertices[cVert++] = m_memberPointers[0]->bottomLeft;
    vertices[cVert++] = m_memberPointers[0]->bottomRight;
    vertices[cVert++] = m_memberPointers[0]->bottomRight;
    vertices[cVert++] = m_memberPointers[0]->topRight;
    vertices[cVert++] = m_memberPointers[0]->topLeft;
    offset += 6;

    for (size_t cMem = 1; cMem < m_memberPointers.size(); cMem++)
    {
        if (m_memberPointers[cMem]->texture != m_memberPointers[cMem - 1]->texture)
        {
            m_renderBatches.emplace_back(offset, 6, m_memberPointers[cMem]->texture);
        }
        else
        {
            m_renderBatches.back().numVertices += 6;
        }
        vertices[cVert++] = m_memberPointers[cMem]->topLeft;
        vertices[cVert++] = m_memberPointers[cMem]->bottomLeft;
        vertices[cVert++] = m_memberPointers[cMem]->bottomRight;
        vertices[cVert++] = m_memberPointers[cMem]->bottomRight;
        vertices[cVert++] = m_memberPointers[cMem]->topRight;
        vertices[cVert++] = m_memberPointers[cMem]->topLeft;
        offset += 6;
    }

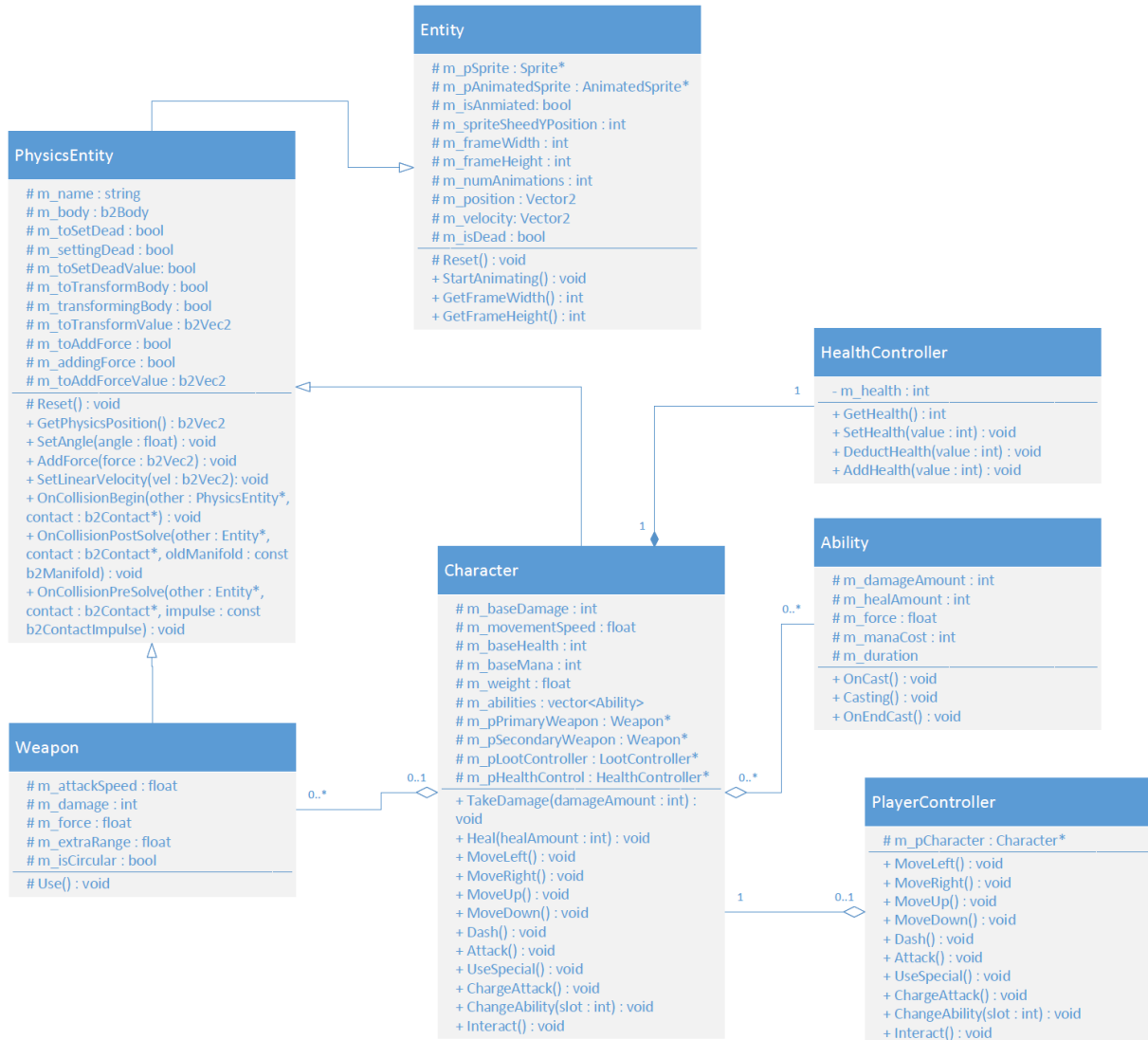
    glBindBuffer(GL_ARRAY_BUFFER, m_vbo);
    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), nullptr, GL_DYNAMIC_DRAW);
    glBufferSubData(GL_ARRAY_BUFFER, 0, vertices.size() * sizeof(Vertex), vertices.data());
}
```

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

```
}
```

```
//Draw a sprite using direction for rotation
void SpriteBatch::draw(const glm::vec4& destRect, const glm::vec4& uvRect,
GLuint texture, float depth, const ColorRGB& color, const glm::vec2& dir)
{
    const glm::vec2 right(1.0f, 0.0f);
    float angle = acos(glm::dot(right, dir));
    if (dir.y < 0.0f)
    {
        angle = -angle;
    }
    m_members.emplace_back(destRect, uvRect, texture, depth, color,
angle);
}
```


UML Diagram (Character)



Character

The base character class will include most of the variables and methods for all the players and enemies. When initializing a character, the difference in objects they have will be the sprite, abilities and weapons. There may not be any need to have classes that extend Character, but there will be a factory that will create the characters with whatever stats they need.

Player Controller

This class handles the input between the user and one of the characters. Essentially, all characters in the game can be playable with how we will setup the Character class.

Health Controller

Every character has a health controller. The health controller stores and updates the health. This is to minimize the length of the name when calling the function. This will separate health functions from characters to make things more simplified.

Weapon controller

Character has a weapon controller. The character will pick up a weapon object which has all the characteristics of that weapon. The weapon controller will handle the use function and make sure the values being passed is accurate to the current weapon values.

Loot Controller

Character has a loot controller. The loot controller manages the functions to do with loot and makes sure the values being used are matching the values of the loot item.

Artificial Intelligence

We will be using the State Design Pattern to handle the AI states of the game. All states will be a singleton object that implements a State interface with the abstract methods of:

```
virtual void Enter(Character& c) = 0;  
virtual void Execute(Character& c, float deltaTime) = 0;  
virtual void Exit(Character& c) = 0;
```

A* Algorithm

The A* Search Algorithm is a path finding algorithm that we will be implementing into our A.I. enemies to navigate to the player's location. The algorithm is the process of finding a path between multiple nodes to decide which path is the shortest from node A to node Z. It will take into account wall collisions. We are using a grid system and setting each grid tile to a node.

Each node has a unique identifier, and a placeholder for parent node.

Local dist - Used to compare nodes to determine if it is the shortest path.

Global dist - measurement of distance between the node and the target node, it will update it through each node test if prompted to.

List - a list of nodes discovered by the algorithm that needs to be tested.

Example:





Starting node (A) will be added to the list, and will check for neighbour node. The neighbour node (B) is found and is added to the list of nodes to test. Node (B) global and local dist is infinity. Current node (A) takes local distance (2) and checks neighbour node (B)'s local distance (infinity). If local distance is smaller than node (B)'s local distance, then update node (B). Node (B) has stored node (A) as parent node. Node (B) updates local distance to be 2, and global distance is the local distance plus the heuristics which is the distance between node (B) and the target node. Update the list by adding the global distance to the respective nodes. Once node (A) discover and visits all neighbouring nodes, it is popped off the list. In the list, each node is sorted in ascending order based on the global distance value, this gives us the best chance of finding the shortest path.

This is repeated until the it has found a path to the target node.

Once it has found a path, it's not certain if that's the shortest path, therefore it will go through the remaining nodes in the list to check if it's path is shorter.

```
if(NODE.local < NEIGHBOUR.Local){  
    Update(NEIGHBOUR, current.NODE);  
}
```

Team Sign-Off

Alvin Tang	
Ian Christopher Apino	
David Tea	
Callum Drennan	

References

(n.d.). Retrieved from <https://glm.g-truc.net/0.9.9/index.html>

(n.d.). Retrieved from <https://www.fmod.com/>

(n.d.). Retrieved from <https://box2d.org/>

About SDL. (n.d.). Retrieved from <https://www.libsdl.org/>

LodePNG. (n.d.). Retrieved from <https://lodev.org/lodepng/>

The OpenGL Extension Wrangler Library. (n.d.). Retrieved from <http://glew.sourceforge.net/>