**Game Engine Architecture - Individual Reflection**

Matthew Cheung - 15010695

**Development Diary**

**7 February 2017**

The player class was added and player class and included it into the game with a sprite. It inherited the ImageGO2D class so that it could render a sprite of some sort.

**7 February 2017 – 14 February 2017 – Creating animated Sprite class**

I also started on an animated sprite class for the player so that it displays a walk animation when it is moving. However, this was not attached to the player yet. I used a followed a tutorial, called [1]Game Tutorial: Sprite Class, part 1: Animating, to implement some sort of animated sprite.

AnimatedSprite::AnimatedSprite(string \_fileName, ID3D11Device\* \_GD, int frameCount): ImageGO2D(\_fileName, \_GD)

{

m\_frameCount = frameCount;

m\_totalElapsed = 0;

m\_frame = 0;

m\_timePerFrame = 0.33;

//Determine how large the image is

ID3D11Resource\* resource;

m\_pTextureRV->GetResource(&resource);

D3D11\_RESOURCE\_DIMENSION dim;

resource->GetType(&dim);

D3D11\_TEXTURE2D\_DESC desc;

((ID3D11Texture2D \*)resource)->GetDesc(&desc);

D3D11\_TEXTURE2D\_DESC desc;

((ID3D11Texture2D \*)resource)->GetDesc(&desc);

m\_frameWidth = desc.Width;

m\_frameHeight = desc.Height;

sourceRect = new RECT();

}

In the code above is the constructor of the Animated Sprite class. Values such as the amount frames in the sprite sheet and other ImageGO2D values. The main purpose of this constructor is find out how big the sprite sheet is and how many frames it contains.

void AnimatedSprite::Draw(DrawData2D\* \_DD)

{

\_DD->m\_Sprites->Draw(m\_pTextureRV, m\_pos, sourceRect, m\_colour, m\_rotation, m\_origin, m\_scale, SpriteEffects\_None);

}

The code above is the draw function. The difference between the ImageGO2D Draw function is that the animated sprite uses a RECT class to “cut out” the desired frame of the sprite sheet and draw that section of it. While ImageGO2D uses the whole image.

Player states have also been added and a “Jumping” action so that the player can now move around the 2D world.

**14 February 2017 – 1 March 2017 – 2D Camera**

Next job was to create a camera that follows the player and a camera which can be moved around to debug the game. Another tutorial I follow is [2] XNA Camera 2d with zoom and rotation. This was mainly used create a base camera class for both the debug camera and a player camera. The debug camera uses the WASD keys to move while in debug mode.

In the code below is the tick function of the base 2D camera which calculates it’s own transform:

void Camera2D::Tick(GameData\* \_GD)

{

m\_transform =

Matrix::CreateTranslation(Vector3(-m\_pos.x, -m\_pos.y, 0)) \*

Matrix::CreateRotationZ(m\_rotation) \*

Matrix::CreateScale(Vector3(m\_zoom, m\_zoom, 1)) \*

Matrix::CreateTranslation(Vector3(\_GD->viewportWidth \* 0.5f, \_GD->viewportHeight \* 0.5f, 0.0f));

GameObject2D::Tick(\_GD);

}

For the debug camera the tick function takes the users inputs and moves it depending on whether the user presses WASD. This can only happen as long as the game is in debug mode;

void DebugCamera::Tick(GameData\* \_GD)

{

if (\_GD->m\_GS == GS\_PLAY\_DEBUG\_CAM)

{

if (\_GD->m\_keyboardState[DIK\_A] & 0x80)

{

m\_pos.x -= 200 \* \_GD->m\_dt;

}

else if (\_GD->m\_keyboardState[DIK\_D] & 0x80)

{

m\_pos.x += 200 \* \_GD->m\_dt;

}

if (\_GD->m\_keyboardState[DIK\_W] & 0x80)

{

m\_pos.y -= 200 \* \_GD->m\_dt;

}

else if (\_GD->m\_keyboardState[DIK\_S] & 0x80)

{

m\_pos.y += 200 \* \_GD->m\_dt;

}

}

Camera2D::Tick(\_GD);

}

And for the player cam:

void CameraFollow2D::Tick(GameData\* \_GD)

{

Matrix rotation = Matrix::CreateFromYawPitchRoll(0, 0, m\_targetObject->GetRot());

m\_target = m\_targetObject->GetPos();

m\_pos = m\_target;

Camera2D::Tick(\_GD);

}

The player cam has an added member of a game object pointer to reference the object the camera needs to follow. In the tick function the position will be updated to point to the target. In the case for this game it is the player sprite.

**1 March 2017 – 20 April 2017 – Applying the flyweight programming pattern**

Currently each class creates a sprite of its own. This is very inefficient as there are some objects in the game which load in the same sprite. So instead a sprite is assigned to the objects and then it moves and draws the sprite in its position and moves on to the next sprite. This means that there are less sprites for the game to load. An example of it is used in the tile class in the code bellow.

class Tile :public GameObject2D

{

public:

Tile(Sprite\*, Vector2);

~Tile();

virtual void Draw(DrawData2D\* \_DD);

Sprite\* getSprite() { return sprite; };

private:

Sprite\* sprite;

};

In the code, there is a sprite pointer used to point at the sprite the object is needed. There is also a get Sprite function to get the dimensions of the sprite collisions. The sprite is then drawn in draw function of the object.

void Tile::Draw(DrawData2D \* \_DD)

{

sprite->SetPos(m\_pos);

sprite->SetScale(m\_scale);

sprite->SetRot(m\_rotation);

sprite->SetColour(m\_colour);

sprite->Draw(\_DD);

}

In the code above is the draw function of tile which has a sprite assigned to it. The draw function simply uses the objects parameters like position, scale, rotation and colour, to print it onto the screen.

**20 April 2017 – 20 April 2017 – Improving the collision system**

Initially the collision system used the distance between the two objects and determine whether they collide or not. To improve the collision system is to implement a bounding box system to determine whether 2 objects collide or now. I applied a bounding box using [4] Bounding Box Collision Detection as reference to how 2 objects can detect collisions using the sprites.

A way that an objects sprite can be accessed is using dynamic casting to access the sprites dimensions to determine the sprites collisions.

void CollisionManager::checkCollision(Room\* room)

{

bool collided = false;

//cycle through all game objects

for each (GameObject2D\* obj in room->getColldingObjects())

{

//if the player collides with collider

if (isCollided(room->getPlayer(), obj))

{

if (room->getPlayer()->isAlive())

{

if ((obj)->GetType() != ObjectType::PLAYER)

{

Direction dir;

dir = checkDirection(room->getPlayer(), obj);

//Do collision behaviour

resolveCollision(room, obj, dir);

collided = true;

}

}

}

}

//if nothing collides

if (!collided)

{

room->getPlayer()->SetIsGrounded(false);

room->getPlayer()->setOnLadder(false);

}

}

In the code above is the main collision this is where it cycles through all the game objects to check whether the player is colliding with anything. If nothing collides then gravity is applied to the player. However, most of the collision behaviours have not changed.

if (dynamic\_cast<Tile\*> (gameObject2) != NULL)

{

Tile\* tile = dynamic\_cast<Tile\*> (gameObject2);

if ((tile->getSprite()->getSpriteHeight() \* tile->getScale().y) + tile->GetPos().y >= player->getSprite()->GetPos().y &&

tile->GetPos().y <= player->GetPos().y + (player->getSprite()->getSpriteHeight() \* player->getScale().y) &&

(tile->getSprite()->getSpriteWidth() \* tile->getScale().x) + tile->GetPos().x >= player->GetPos().x &&

tile->GetPos().x <= (player->getSprite()->getSpriteWidth() \* player->getScale().x) + player->GetPos().x)

{

return true;

}

In the code above is how the game detects collisions. It uses the bounding model with the sprite sizes from the object.

Directional collisions were also added using [5] as a reference. An example of how the game uses it is in the code below:

if (dynamic\_cast<Tile\*> (obj2) != NULL)

{

Tile\* tile = dynamic\_cast<Tile\*> (obj2);

//getting sprite sizes

float playerBot = player->GetPos().y + (player->getSprite()->getSpriteHeight() \* player->getScale().y);

float playerRight = player->GetPos().x + (player->getSprite()->getSpriteWidth() \* player->getScale().x);

float platBot = tile->GetPos().y + (tile->getSprite()->getSpriteHeight() \* tile->getScale().y);

float platRight = tile->GetPos().x + (tile->getSprite()->getSpriteWidth() \* tile->getScale().x);

//calculate collisions

float botCollision = platBot - player->GetPos().y;

float topCollision = playerBot - tile->GetPos().y;

float leftCollision = playerRight - tile->GetPos().x;

float rightCollision = platRight - player->GetPos().x;

//Top Collision

if (topCollision < botCollision &&

topCollision < leftCollision &&

topCollision < rightCollision)

{

return Direction::TOP;

}

The code uses both sprites and compares the position of each side to determine which side the player is colliding from.

**Post Mortem**

**Goals**

The main goal of the team was to create a game engine which can run a game with similar functionality of the game Jet Set Willy and a basic platformer.

**Successes**

Group management

The team overall had very good management to allocate what each team member did in terms of what we were good at. We also had a backlog to keep other team members up to date with the progress of the features that each team member oversaw. The team also had meeting regularly to also update on the progress of these features as well.

**Failures and Issues**

Time Management

Time management was one of the issues which we had to tackle as a team. This is because we had a 3-week gap where little or no progress was made with the game. One way to solve this issue was to

Coding Style

One if the issues we had was to keep a consistent coding style throughout the codebase of the game engine. Even though most of the code is still readable with differing coding standards from all team members, this could’ve caused some confusion while other team members read the code in the game engine. A solution to this was to establish a coding standard before the implementation of the game started.

**Lessons Learnt**

One of lessons learnt during this project is how important planning is before implementing game features. Throughout the implementation of the game the team just went into it and started implementing the features without clearly planning the implementation. With this planning can lead to more optimised code and also cleaner code in general as currently the code in the games implementation is not very efficient in some places which could be improved immensely.

**References**

*[1]Sobolewski P. (2015) Game Tutorial: Sprite Class, part 1: Animating. Available form:* <http://patsobo.com/game-tutorial-sprite-class-part-1/> *[Accessed 7 February 2017]*

*[2] Amador D. (2009) XNA Camera 2d with zoom and rotation. Available from:* <http://www.david-amador.com/2009/10/xna-camera-2d-with-zoom-and-rotation/> *[Accessed 14 February 2017]*

*[3] Nystrom R. (2014) Game Programming Patterns [online]. Kindle ed. Genever Benning*

*[4]* Atrixium. (2012) Bounding Box Collision Detection. YouTube [video]. 20 October. Available from: <https://www.youtube.com/watch?v=8b_reDI7iPM> *[Accessed 20 April 2017]*

*[5]* <http://stackoverflow.com/questions/5062833/detecting-the-direction-of-a-collision>