Games Programming and System Architecture

CMP 208 Project Report

Utkarsh Yadav / 1905406

Thomas had a gun

Fig - 1 GameScreen

Introduction

Thomas had a gun is a first-person shooter game inspired by the indie game Thomas was alone. The main objective of the game is to kill or shoot all the enemies. If the player is within the enemy range the enemy moves towards the player and tries to kill him. The player has a gun to defend themself. At the start of the game the player is given the option to choose between two guns, both of the guns can kill enemies, but they have their own perks, one is a shotgun and the other is an ar(assault rifle). The game arena is populated with trees and has a base object which acts as a playing ground. The base arena is

surrounded by invisible walls so that the player and the enemies do not leave the map. Killing all the enemies wins the game if the player dies they lose.

User guide

Controls

- Up and Down keys to browse the menu and Enter to select something in the menu ESC to go back.
- WASD to move around.
- Space to shoot bullets.
- Mouse movement to rotate the player.
- Esc to pause the game.
- To quit the game press ESC then exit.

Application design

Game Arena

To make an fps with 2d collisions the most essential thing is to know which axis to use for the game. In this particular case, the x-z plane is being utilised to do all the movements and collision, the Y-axis remains constant so that the game draws objects and bodies horizontally and not vertically. Because the x and z plane are being used the gravity in the game is set to zero so that it doesn't interfere with player and enemy movements. Neither the player nor the enemies can leave the playing arena as there are 4 static b2Bodies surrounding the playing ground the static bodies do not have any mesh assigned to them so it seems like there are invisible walls surrounding the arena. The Game Arena is a 50*0.5*50 cube with grass textures.

Player and Enemy movements

Both the player and enemy bodies translate or move in the x-z plane, the player body can move in all 4 directions using user input, on the other hand, the enemy bodies will stay still until the player is in range, once the player is in range they will move towards the player and if they are close enough to the player, the player loses a certain amount of health. If the player doesn't move away from the enemies they'll lose all their health and also lose the game. To win the game the player has to shoot and kill all the enemies. The shooting mechanic is explained in the next section.

Bullets and Guns

The second most essential thing in an fps(first-person shooter) is the shooting mechanic. There are two different types of guns in the game, ar(assault rifle) which shoots one single bullet body but has a faster rate of fire, the other gun is a shotgun that shoots two bullet bodies but has a slower rate of fire. The player is given the options to choose between the two at the start of the game. Each bullet is initialised and pushed into a bullet vector which is rendered in the Gamerender function.

```
bullet.Init();

if (gunType == "ar")
{
        audio_manager_->PlaySample(3, false);
        timer_ = 0.1f;
        bullet.setPosition(bullet_pos, player_Rotation_);
        bullet.shoot(80, gunType, camera_fwd);
        bullets_.push_back(bullet);
}
```

Fig - 2 shooting bullets

the above code shows how every bullet is initialised before pushing into the bullets_ vector. The same code is used for initialising the shotgun bullets. The timer_ variable is used for the variable rate of fire.

```
if (keyboardManager->IsKeyDown(gef::Keyboard::KC_SPACE) && timer_ <= 0.f)
```

Fig - 3 If statement

```
timer_ -= frame_time;
```

Fig - 4 timer decrements

The if statement checks if the space key is being pressed and the timer_ variable is lower than equals to zero, to change the rate of fire for the bullets set the timer_ variable to anything between 0 to 1 for the AR gun it is 0.1 and for the shotgun, it is 0.5. At the end of the if loop the timer variable gets decremented by frame_time.

Trees

The game arena is populated with tree meshes that are equidistance to each other. All the trees meshes are part of a mesh instance array the array size can be changed to add or remove trees from the game. The trees do not have their own b2 bodies so players and enemies are able to move through them. The only reason it is like this is to reduce the number of collisions happening in the game.

Fig - 5 Tree model

Techniques Used

Camera Class

The camera class functions as the player as it is an fps(first-person shooter). The camera only moves in the x and z plane and is constant on the y-axis as all the collisions are handled by a 2d engine. The camera or the player in this case has its own b2Bodies, one is the player body the other is a player boundary which is a censor body used for the enemy AI if the enemies collide with that body the enemies move towards the player and damage the player that's where the player b2Body comes into action. When the player body collides with the enemy body the player's health gets reduced by a certain amount. The camera

class also sets the position for the gun, the position for the guns are implemented as follows:-

```
shotgunposition = position + gef::Vector4(0.0f, -1.8f, 0.f) + (forward * 1.8f) + (right * 0.1f);
gef::Matrix44 guntransform, guntranslation, gunrotation;
gunrotation.SetIdentity();
gunrotation.RotationY(-300.f + gef::DegToRad(-Yaw));
guntranslation.SetIdentity();
guntranslation.SetTranslation(shotgunposition);

shotgun.set_transform(gunrotation * guntranslation);
```

Shotgunposition is a vector4 which is initialised with camera position vector, camera forward vector and camera right vector. The camera position is 1.8 units above the ground so that the player is able to see the gun when shooting. The forward and right vectors are used for proper gun rotation. If they aren't used, the gun rotates on a single point and doesn't rotate with the camera. All the 4x4 Matrixes are used for rotations and translation for the gun. It also does it for both guns.

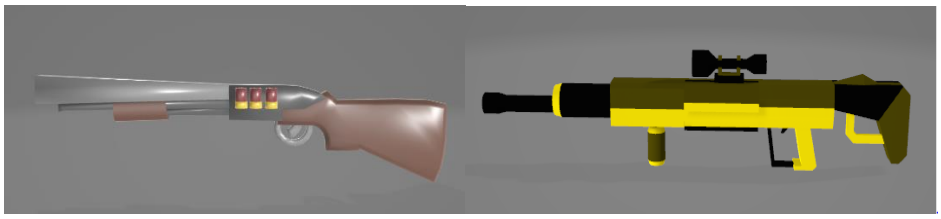

Fig - 6 shotgun and ar gun models

The movement of the player is handled by the forward and right vector they are initialised using trigonometric functions. All the calculations are done in the update function of the camera class here's a snippet of the update function.

```
float cosR, cosP, cosY;  //temp values for sin/cos from
float sinR, sinP, sinY;

// Roll, Pitch and Yallare variables stored by the camera
// handle rotation
// Only want to calculate these values once, when rotation changes, not every frame.
cosY = cosf(Yaw * 3.1415 / 180);
cosP = cosf(Pitch * 3.1415 / 180);
cosR = cosf(Roll * 3.1415 / 180);
sinY = sinf(Yaw * 3.1415 / 180);
sinP = sinf(Pitch * 3.1415 / 180);
sinR = sinf(Roll * 3.1415 / 180);

//This using the parametric equation of a sphere
// Calculate the three vectors to put into glu Lookat
// Look direction, position and the up vector
// This function could also calculate the right vector
forward.set_x(sinY * cosP);
forward.set_y(0.f);
//forward.set_y(sinP);
forward.set_z(cosP * -cosY);



// Look At Point
// To calculate add Forward Vector to Camera position.

// Up Vector
up.set_value(0.f, 1.f, 0.f);


// Side Vector (right)
// this is a cross product between the forward and up vector.
// If you don't need to calculate this, don't do it.
right = forward.CrossProduct(up);

right.Normalise();
LookAt = position + forward;
```

Fig - 7 Update function camera class

The up vector and forward set_y is kept constant as not to move in that direction at all.

Everything else is just basic camera movement techniques used in many other frameworks like OpenGL and unity.

The handle input function inside the camera function updates the vector if any movement key is pressed.

```
if (keyboardManager)
{
    if (keyboardManager->IsKeyDown(gef::Keyboard::KC_A))
    {
        newpos += right * (-mov_speed * dt);
    }
    if (keyboardManager->IsKeyDown(gef::Keyboard::KC_D))
    {

        newpos += right * (mov_speed * dt);
    }
    if (keyboardManager->IsKeyDown(gef::Keyboard::KC_W))
    {

        newpos += forward * (mov_speed * dt);
    }
    if (keyboardManager->IsKeyDown(gef::Keyboard::KC_S))
    {

        newpos += forward * (-mov_speed * dt);
    }

    player_body->SetLinearVelocity({newpos.x(), newpos.z()});
    position.set_x(player_body->GetPosition().x);
    position.set_z(player_body->GetPosition().y);
}

if (touch_input)
{
    ShowCursor(false);
    mousex = touch_input->mouse_rel().x();

    Yaw += mousex * 0.08f;
}
```

Fig - 8 handle input function camera class

Whenever the player presses any of the movement keys the forward and the right vector gets multiplied by movement speed and gets incremented into the newpos which is a vector4. Newpos vector is being used to set the linear velocity of the player body and the player body position is being used to set the camera position. Yaw the only rotation variable used as the game only needs rotation on the x-axis.

Projectiles Class

The projectile class handles all the bullet functionality in the game, it sets the bullet body position to where the player is at a point in the game, it also shoots the bullet body from the position and the forward vector of the player, even if the player rotates the bullet gets shot in front of him. The projectile class has two b2Bodies one is a normal bullet body and the other one is an array of two bullet bodies the code used is as follows:-

```
GameObject projectile_;
b2Body* projectile_body_;
b2Body* shotgun_bullet_body_[2];
GameObject shotgun_bullet_[2];
```

fig - 9 bullet body declarations

The Gameobject class is used to update the set meshes with their b2Bodies. The initialisation of the b2Bodies is done in their respective functions. Here is one for example

```
for (int i = 0; i < 2; i++)
{

    gef::Vector4 projectile_half_dimensions(0.05f, 0.05f, 0.05f);
    shotgun_bullet_[i].set_mesh(primitive_builder_->CreateBoxMesh(projectile_half_dimensions));

    b2BodyUserData user_data_;
    user_data_.pointer = 5;
    //create a physics body for the projectiles
    b2BodyDef projectile_body_def;
    projectile_body_def.type = b2_dynamicBody;
    projectile_body_def.userData = user_data_;
    projectile_body_def.bullet = true;
    projectile_body_def.position = b2Vec2(1000+i , -5.0f);

    shotgun_bullet_body_[i] = world_->CreateBody(&projectile_body_def);

    //create the shape for the projectile
    b2PolygonShape projectile_shape;
    projectile_shape.SetAsBox(0.05f, 0.05f);

    // Create Ficture
    b2FixtureDef projectile_fixture_def;
    projectile_fixture_def.shape = &projectile_shape;
    projectile_fixture_def.density = 1.0f;
    // Create the fixture on the rigid body
    shotgun_bullet_body_[i]->CreateFixture(&projectile_fixture_def);

    //update visuals from simulation data
    shotgun_bullet_[i].UpdateFromSimulation(shotgun_bullet_body_[i]);
    shotgun_bullet_[i].y_pos = 1.5f;

}
```

fig - 10 shotgun body init

As the shotgun body is an array of two it's in a for loop to get initialised twice, the same is done for the projectile body, the bodies are also initialised with a b2BodyUserdata to help with collision detection and collision filtering. The shooting mechanic is a two-part process the first part is to set the bullet position to the player position, second is to shoot the bullet using Linear Impulse to the centre. Here is a snippet of the code used for achieving that:-

```
projectile_body_->SetTransform(position,rotation.GetAngle());
projectile_.UpdateFromSimulation(projectile_body_);
```

Fig 11 updating position

```
projectile_body_->ApplyLinearImpulseToCenter(position, true);
projectile_.UpdateFromSimulation(projectile_body_);
```

Fig 12 shooting the bullet

The position in transform is the camera position and the position in apply impulse is the camera forward vector multiplied by the velocity of the bullet.

GameObject Class

The function of the game object class is to update the meshes used in the game with their respective b2Bodies. The update from simulation function in the class works as follows:-

```
// setup object rotation
gef::Matrix44 object_rotation;
object_rotation.RotationY(body->GetAngle());

// setup the object translation
gef::Vector4 object_translation(body->GetPosition().x, y_pos, body->GetPosition().y);

// build object transformation matrix
gef::Matrix44 object_transform = object_rotation;
object_transform.SetTranslation(object_translation);
set_transform(object_transform);
```

Fig 13 UpdatefromSimulation function

Instead of Rotation along the z-axis, this function rotates all the bodies in the y-axis also the translation is being affected on and x and z axes and the y axis is a constant for each specific body. That is how the game achieves proper 2d collisions in a 3d environment.

Enemy Class

The enemy class has a single Game object and b2body which is then pushed into a vector to spawn multiple enemies, each enemy is initialised with a designated size and material. The movement of all the enemies is handled in the collision response function in scene class. The enemy also has a b2Body function which returns a b2Body this helps in enemy removal. The set and get health functions are used to specify how many bullets will it take to kill an enemy its different for both the guns

Fig 14 enemy body with textures

SceneApp Class

The SceneApp class is the most important class in the game as it initialises, updates and renders everything in the game. All the other classes and objects are initialised in the Init function of the SceneApp class. All the collision and collision responses are handled in this class.

CollisionDetection function

The collision detection function uses the b2Contact variable to get all the contacts happening in the game and iterates through them to check for collision between specific bodies. Each and every b2Body used in the game has its own b2BodyUserData which helps distinguish between bodies while doing collision responses.

If the bullet collides with the walls of the arena the specific bullet body which is colliding with the wall gets pushed into a vector of b2Body and are removed after the collision detection function is done executing. Code snippet for reference:-

```
if ((budA.pointer == 10 && budB.pointer == 5) || (budA.pointer == 5 && budB.pointer == 10))
{
    if (budA.pointer == 5)
    {
        remove_bullets.push_back(bA);
    }
    if (budB.pointer == 5)
    {
        remove_bullets.push_back(bB);

    }
}
```

Fig 15 bullet removal after collision with the wall

budA and budB are body user data for the respective bodies, it checks if either of the two bodies is a bullet or wall body if they are it pushes that body bA or bB (body A or body B) into the vectors of bodies. The same is done if the enemy body collides with the bullet body the enemy body and the bullet body gets pushed into the vector of b2Bodies and are deleted or removed in the remove bodies function.

If the player boundary body collides with the enemy body it moves towards the player body and if the player body collides with the enemy body the player loses health. Code snippet for reference:-

```
b2Vec2 velocity = Player_body_->GetPosition() - bA->GetPosition();
velocity.Normalize();
velocity *= 2.f;
bA->SetLinearVelocity(velocity);
```

Fig 16 moving the enemy body towards the player

The enemy moves towards the player using the position of the player subtracted by the enemy body position multiplied by a constant speed.

```
if (budA.pointer == 3 && budB.pointer == 22 || budA.pointer == 22 && budB.pointer == 3) {
    health -= 1;
}
```

Fig 17 players health getting reduced if enemy body collides with it

The player's health is decremented by one if it collides with the enemy body or the enemy body collides with the player.

Remove_bodies function

The remove body function removes all the bodies that have been pushed into the remove body vectors the way it does that is as follows:-

```
for (size_t i = 0; i < remove_enemies.size(); i++) {
    for (size_t j = 0; j <enemies_.size(); j++) {
        if (enemies_[j].getBody() == remove_enemies[i]) {
            int health = enemies_[j].getHealth() - 1;
            enemies_[j].setHealth(health);

            if (enemies_[j].getHealth() <= 0) {
                enemies_[j].getBody()->GetWorld()->DestroyBody(enemies_[j].getBody());
                enemies_.erase(enemies_.begin() + j);
            }
        }

    }
}
```

Fig 18 removing enemy bodies

The for loop iterates through all the bodies that are bound to be removed and check those bodies with the existing enemy body in the scene, if they match the body gets deleted but not until the health of the enemy is zero one bullet hit decrements health by one. The same is done for the bullet bodies, except they don't have any health so they get removed instantly.

GameState Class

The game state class has an enum called States which has nine different states in it. It also has two function getState and setState which returns and sets current state respectively.

To use the game state class there is an update state function inside the SceneApp class which has a switch statement for each state and respective update and render functions. Here is a snippet of how it looks like:-

```
switch (game_state_.getCurrentState()) {
case State::Init:
    InitStateUpdate(dt);
    InitStateRender();
    break;
case State::CONTROLS:
    ControlsStateUpdate(dt);
    ControlsStateRender();
    break;
case State::OPTIONS:
    OptionsStateUpdate(dt);
    OptionsStateRender();
    break;
case State::LEVEL:
    GameUpdate(dt);
    GameRender();
    break;
case State::PAUSE:
    PauseStateRender();
    PauseStateUpdate(dt);
    break;
case State::PAUSE_CONTROLS:
    ControlPauseStateupdate(dt);
    ControlPauseStaterender();
    break;
case State::DEATH:
    DeathStateUpdate(dt);
    DeathStateRender();
    break;
case State::CREDITS:
    CreditStateUpdate(dt);
    CreditStateRender();
    break;
```

Fig 19 GameState Update function

Audio and Graphics

All the audio and textures of the game are loaded in the SceneApp Init functions are used at their respective places

```
//loading all the game sounds
audio_manager_->LoadMusic("faded_menu.wav", platform_);
audio_manager_->LoadSample("browsing.wav", platform_);
audio_manager_->LoadSample("enter_.wav", platform_);
audio_manager_->LoadSample("shotgun.wav", platform_);
audio_manager_->LoadSample("ar_one_bullet.wav", platform_);
audio_manager_->LoadSample("walking.wav", platform_);
```

Fig 20 loading audios

All the audio used in the game is loaded through the audio manager the game has one music file and multiple sample files which play when specific conditions are satisfied.

```
//loading game textures
mainscreen = CreateTextureFromPNG("TitleScreen.png", platform_);
controlscreen = CreateTextureFromPNG("ControlsScreen.png", platform_);
optionscreen = CreateTextureFromPNG("OptionsScreen.png", platform_);
grass_ = CreateTextureFromPNG("grass.png", platform_);
whompTexture_ = CreateTextureFromPNG("whomp.png", platform_);
winscreen = CreateTextureFromPNG("win_screen.png", platform_);
deathscreen = CreateTextureFromPNG("death_screen.png", platform_);
```

Fig 21 loading textures for sprite and game screens

Each Texture object has its own png file assigned to it and it uses gef::sprite to either render the texture on screen or bound the texture to a material.

Data-Oriented Design

The number of entities used in the game is far too low for a data-oriented design, using a data-oriented design would be Detrimental to the clearness of the code. For example, if the game was using more than a thousand bullet bodies at any given point in the game it would make sense to use a more data-oriented design than what it's using now. The same goes for the enemies. If the game had more bullets using fewer pointers and heap allocations would be a better option than the current one.

Reflection

Working with box2d was a delight, I cannot say the same for gef. Gef seems to be a little unpolished in a few places such as how you have to make two different matrixes for rotation and translation. But it was still a fun experience, this was my first 3d FPS game and getting my head around using a 2d collision framework in a 3d environment took a little time but when I understood how that works everything else was much easier than I anticipated thanks to box2d. I would definitely use box2d in the future it seems like a very well designed framework. It was a shame that we didn't get to play around with the ps vita, I would've loved to see my game work on a ps vita. Other than that I really enjoyed this module learned a lot of things about the 3d aspects of games.

References

2021. [online] Available at: <https://twitter.com/mariobrothblog/status/882710574471294977?lang=en> [Accessed 20 May 2021].

DownloadFree3D.com. 2021. *Low poly shot-gun | DownloadFree3D.com*. [online] Available at: <https://downloadfree3d.com/3d-models/weapons/military-weapons/low-poly-shot-gun/> [Accessed 20 May 2021].

Drink, F., Design, I., Models, F., Models, 3., Models, M., Models, C., Models, B., Models, L., Models, A., Models, R., Models, O., Models, F. and Tree, L., 2021. *free tree 3d model*. [online] Turbosquid.com. Available at: <https://www.turbosquid.com/3d-models/free-tree-3d-model/941297> [Accessed 20 May 2021].

Fesliyanstudios.com. 2021. *Free Gun Shooting Sound Effects | MP3 Download | FStudios*. [online] Available at: <https://www.fesliyanstudios.com/royalty-free-sound-effects-download/gun-shooting-300> [Accessed 20 May 2021].

Koenig, M., 2021. *Mossberg 500 Pump Shotgun Sounds | Effects | Sound Bites | Sound Clips from SoundBible.com*. [online] Soundbible.com. Available at: <https://soundbible.com/2095-Mossberg-500-Pump-Shotgun.html> [Accessed 20 May 2021].

NetrinoMedia, I., 2021. *Guns Free 3D Model in Rifle 3DExport*. [online] 3DExport. Available at: <https://3dexport.com/free-3dmodel-guns-337138.htm> [Accessed 20 May 2021].

SoundCloud. 2021. *faded (out on spotify)*. [online] Available at: <https://soundcloud.com/magotox/faded> [Accessed 20 May 2021].