CMP-208 Coursework Report

Definitely Not Bubble Bobble

Introduction:

"Definitely Not Bubble Bobble" is a shooter-platformer, vastly inspired by the original arcade game "Bubble Bobble". Using characters and similar gameplay mechanics, this game creates a new fun experience for the player. The player plays as a green little dinosaur, fighting against enemies using bubbles. To kill enemies, the player has to trap them in bubbles and then pop them. As soon as an enemy dies, a cherry is spawned in their place, which when collected increases the player's score. The player's objective is to get as higher score as possible. However, surviving becomes progressively harder the more the enemies the player kills. When all enemies of a wave are killed, the number of enemies for the next wave are incremented. This makes the game more challenging the longer the player survives. Moreover, there are three types of enemies, which the game randomly spawns each time, making each playthrough different. When the player loses all three of their hearts, the game ends and they are presented with their score. The score can be displayed as red, yellow, or green exhibiting if its low, medium, or high respectively. The player then has the choice of re-playing the game, starting back from wave one. This game has some customizable features which the player can change depending on their preference, for example the volume of the music and sound effects and their input method (controller or keyboard).

Application Design:

In the game, there are 6 game states. Each state inherits from the generic "State" class (Figure 1) which includes virtual functions that each state overrides to their own needs and pointers to all necessary resources (Platform, Primitive Builder, Sprite Renderer, Input Manager, Renderer 3D, Font, Audio Manager, and the State Machine). The State Machine class is in charge of holding all states and point to the currently active state. All States also have a pointer to the "State Machine" class, so that they can change the current game state when certain conditions are met, using the "changeGameState" function (Figure 2).

```
//Generic State Class
class State
{
public:
	//Initialise
	virtual void Init(gef::Platform* platform,
		PrimitiveBuilder* primitive_builder,
		gef::SpriteRenderer* sprite_renderer,
		gef::InputManager* input_manager,
		gef::Renderer3D* renderer_3d,
		gef::AudioManager* audio_manager,
		const gef::SonyController* controller,
		StateMachine* state_machine) {};
	//CleanUp
	virtual void CleanUp();
	//Update
	virtual void Update(float dt) {};
	//Render
	virtual void Render() {};
	//On Enter Exit
	virtual void OnEnter() {};
	virtual void onExit() {};

protected:
	//Load Asseds
	gef::Mesh* LoadAsset(const char* asset_filename);
	gef::Scene* LoadSceneAssets(gef::Platform& platform, const
char* filename);
	gef::Mesh* GetMeshFromSceneAssets(gef::Scene* scene);

	//Font
	void InitFont();
	void CleanUpFont();

	//Pointers
	const gef::SonyController* controller_;
	gef::Platform* platform_;
	PrimitiveBuilder* primitive_builder_;
	gef::SpriteRenderer* sprite_renderer_;
	gef::InputManager* input_manager_;
	gef::Renderer3D* renderer_3d_;
	gef::Font* font_;
	gef::AudioManager* audio_manager_;
	StateMachine* state_machine_;
};
```

Figure 1- Generic State Class

Every Game State overrides these functions depending on its needs. Since these functions are part of the State class, the current state pointer, in the State Machine class, can be of type State. If any of these functions are called using the State pointer the overridden version will be executed.

These Functions are needed in all State classes in order to Load 3D Assets and initialise the Font.

Pointers to essential resources that all Game States need

When the Game State is changed, the current Game State's "onExit" function is called. (The only exception is when the Pause State is next, since the Level State has to be paused instead of being reset.) Then, when the current state is changed, the new Game State's "onEnter" function is called.

```
void StateMachine::changeGameState(GAME_STATE next_stage)
{
	if (!(current_state == &level_state && (next_stage
== PAUSE)))
	{
		current_state->onExit();
	}
	switch (next_stage)
	{
	case MENU:
		current_state = &main_menu_state;
		break;

	case LEVEL:
		current_state = &level_state;
		break;

	case CONTROLS:
		current_state = &controls_state;
		break;

	case PAUSE:
		current_state = &pause_state;
		break;

	case DEATH:
		death_state.SetScore(score);
		current_state = &death_state;
		break;

	case OPTIONS:
		current_state = &options_state;
		break;
	current_state->OnEnter();
```

Figure 2- Change Game State Function

Figure 3-Main Menu State

Main Menu State:

The Main Menu State (Figure 3) is the first state presented to the player as soon as the game launches. It displays the name of the game and includes three choices: “Options”, “Play”, “Controls”. The player can choose to immediately play the game, learn the controls, or even change the music and sound effects volume. The game state changes depending on the players choice.

Options State:

The Options State (Figure 4) can only be accessed through the Main Menu. It allows the player to change the volume for the game’s music and sound effects. From this state the game can only move back to the Main Menu state.

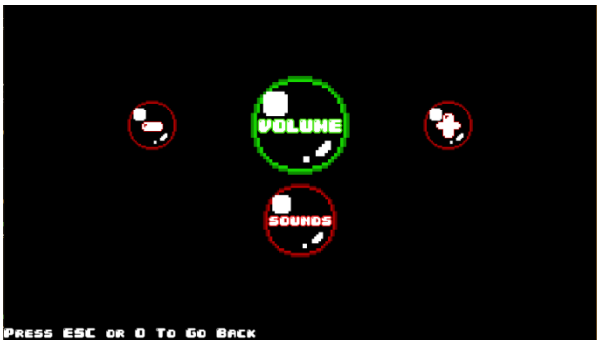


Figure 4-Options State

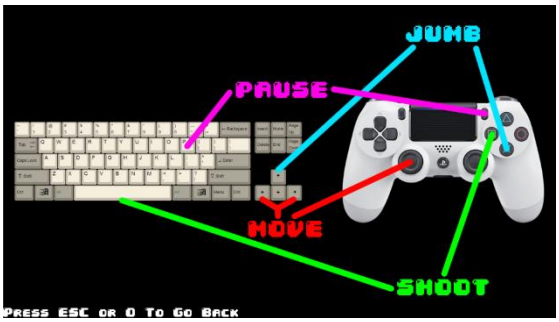


Figure 5-Controls State

Controls State:

The Controls State (Figure 5) is only accessible through the Main Menu. It displays to the player the basic controls of the game for both the keyboard and controller. From this state the game can only move back to the Main Menu state.

Level State:

The Level State (Figure 6) can be accessed when choosing “Play” from the Main Menu. This is the state where the player can play the game and try to get as higher score as possible before losing all their hearts. From this state, the Pause state and the Death state is accessible.

Figure 6-Level State

Figure 7-Pause State

Pause State:

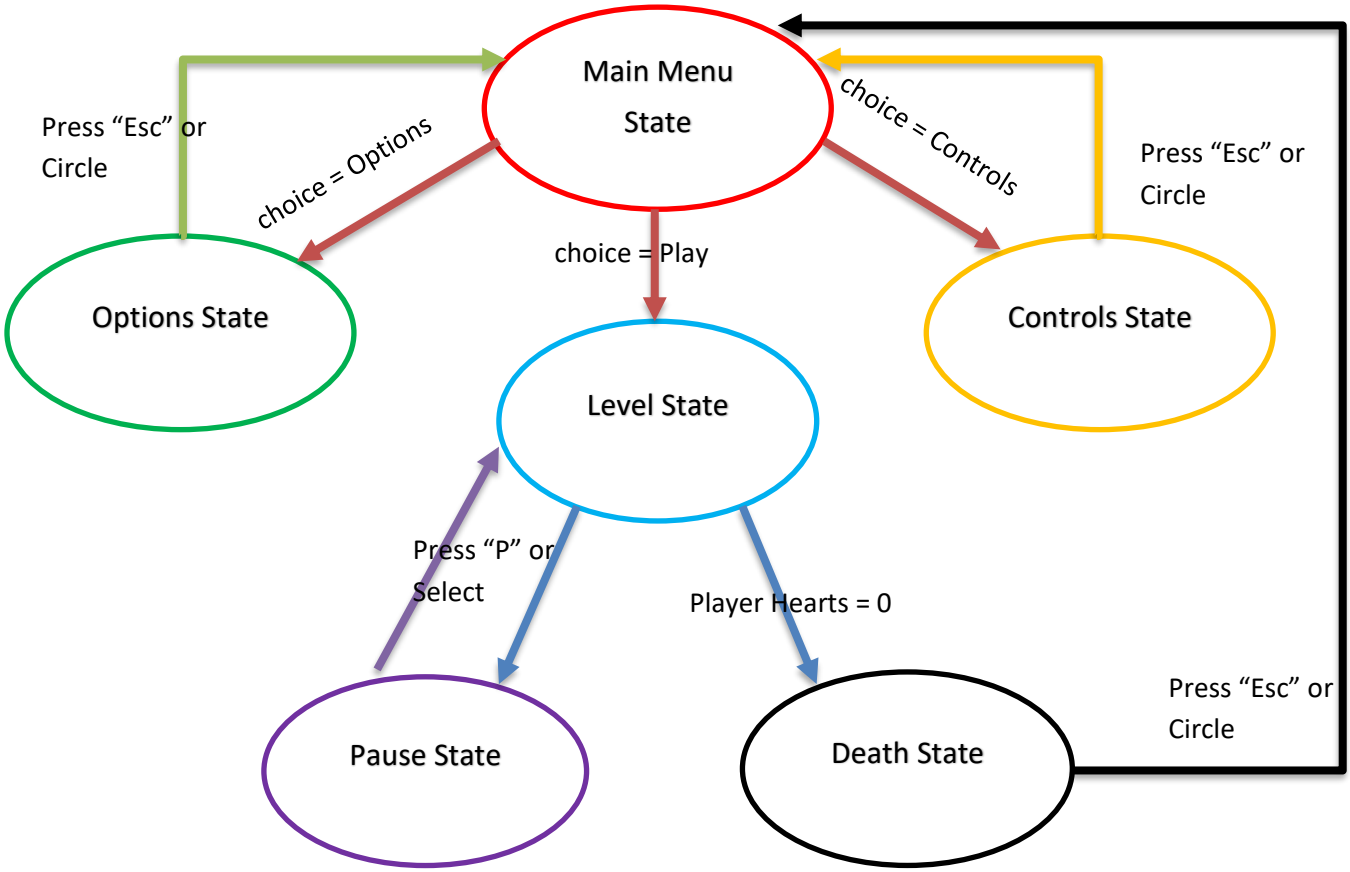
The Pause State (Figure 7) is only accessible while the player is in the level state. When entering this state, the level state remains frozen until the player returns back to it. This state does not do anything except pausing the level state.

Death State:

The Death State (Figure 8) is presented when the player, in the level state, loses all their hearts. This signals the end of the game and displays the final score to the player. From this state the player can return to the Main Menu and re-start the game from the beginning.

Figure 8-Death State

Finite State Diagram of "Definitely Not Bubble Bobble":



Techniques Used:

Game Objects:

Game Objects are an essential part of “Definitely Not Bubble Bobble”. In order to efficiently implement them, a “Game Object” class was created. This class inherits from the Mesh Instance class and has a pointer to its own body. It also includes some common functions needed by all game objects, like “BuildTransformationMatrix” (Figure 9) and “LoopAround” (Figure 10), and some attributes like “object_size” and “rotationY”.

```
void GameObject::BuildTransformationMatrix()
{
	if (body)
	{
		// setup object rotation
		gef::Matrix44 object_rotation;
		object_rotation.RotationZ(body->GetAngle());

		// setup the object translation
		gef::Vector4 object_translation(body->GetPosition().x,
body->GetPosition().y, 0.0f);

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

Figure 9- "BuildTransformationMatrix" function

The “BuildTransformationMatrix” function builds a transformation matrix using the game object’s body’s position and rotation and applies it to the game object’s mesh. It is called when the object is initialised and updated.

The “LoopAround” function checks if the object’s position is lower than the minimum y-axis value of the level or higher than the maximum y-axis value of the level. If this is true, the object is translated to the maximum or minimum y-axis positions respectively.

```
//Loop Around When Falling Off Stage
void GameObject::LoopAround(float maxY, float minY)
{
	//If go out on top
	if (body->GetPosition().y > maxY + object_size.y)
	{
		//Go To Botom
		float translation = minY - object_size.y;
		body->SetTransform({ body->GetPosition().x,translation }, 0);
	}

	//if fall from bottom
	if (body->GetPosition().y < minY - object_size.y)
	{
		//Go To TOP
		float translation = maxY + object_size.y;
		body->SetTransform({ body->GetPosition().x,translation }, 0);
	}

}
```

Figure 10-"LoopAround" function

Moreover, there are five different types of objects (Figure 11) : “Player”, “Enemy”, “Bubble”, “Block” and “Fruit”. Depending on each objects functionality, they are assigned the respective type.

```
enum OBJECT_TYPE
{
	PLAYER,
	ENEMY,
	BUBBLE,
	BLOCK,
	FRUIT
};
```

Figure 11-Object Types

The “Character” Class:

For the Player and Enemy object types a “Character” class (Figure 12), inheriting from the “GameObject” class, was created. It includes an Initialisation function that creates a circle or a square body for the object at a given position, a boolean holding information about the direction of the character and an animated mesh which handles the characters animation.

```
class Character: public GameObject
{
public:
	//Initialise
	void Init(b2World* world, gef::Vector2 position, bool circle);
	//Direction
	bool lookRight = true;
	//Animation
	AnimatedMesh animation;
};
```

Figure 12-"Character" Class

The Player:

For implementing the green dinosaur, a “Player” class was created, inheriting from the Character class. The essential variables included in this class are health, score, shooting cooldown and an immunity counter. Initially, the health is set to three and the score is set to zero. The shooting cooldown is used to make sure the player can’t spam bubbles and the immunity counter is used when the player has been hit to give them a grace period.

The player can move using the functions walk and jump. When called, the jump function checks whether the player is already jumping (Figure 13) , if they don’t their vertical velocity is set to a positive amount and the function returns true.

```
bool Player::jumb()
{
	if (!isJumbing)//If is not jumbing
	{
		body->SetLinearVelocity({ 0,20});

		isJumbing = true;
		return true;
	}
	return false;
}
```

Figure 13-The Jumping Function

The walk function also takes an argument of the direction using a boolean. Depending on that boolean the walking velocity is set a negative or positive value (Figure 14) .

```
void Player::walk(bool right, float dt)
{
	float walk_speed = 400 * dt;
	if (right) //Go Right
	{
		lookRight = true;
		// Walk And Walk Animation
		body->SetLinearVelocity({ walk_speed,body->GetLinearVelocity().y });
	}
	else //Go Left
	{
		lookRight = false;
		//Walk And Walk Animation
		body->SetLinearVelocity({ -walk_speed,body->GetLinearVelocity().y });
	}
	animation.RotateY(lookRight);
	set_mesh(animation.mesh());
}
```

Figure 14-The Walking Function

Shooting a bubble is done using the fire function. If the cooldown is over, this function resets it and sets the mesh of the dinosaur as the firing mesh (Figure 15). It also returns a boolean which is true if the player was able to shoot.

```
bool Player::fire()
{
	if (cooldown >= 1)//If cooldown is over
	{
		cooldown = 0;
		set_mesh(dino_fire.mesh());
		return true;
	}
	return false;
}
```

Figure 15- The Shooting Function

The player's update function is: updating the animation, building a transformation matrix and applying it to the player's mesh, making sure the player didn't fall of the level, incrementing the shooting and immunity cooldowns and checking the player's y-axis velocity to allow the player to jump only when they are touching the ground.
Other important functions include:

- The render function for changing the player's colour when they have immunity.
- The Increasing Score function, which increases the score if the player collects a cherry.
- The Decreasing Health function which decrements the player's health and resets the immunity counter if they don't possess immunity already.

Enemies:

In the game, there are three types of enemies. Each enemy has their own unique movements, but they all share some similarities. Therefore, the Enemy class was created, inheriting from the Character class. When colliding with a bubble, all enemies are required to freeze and move with the bubble. This was achieved using a boolean, signalling if the enemy is captured and a Bubble pointer which is set to the colliding bubble when the collision occurs (Figure 16). When the enemy is later updated, if the captured boolean is positive, the enemies physics body is disabled, and it transforms to the bubbles position creating the illusion that the enemy is trapped inside the bubble. If the enemy is not captured, its Animation, Rotation and Position is Updated.

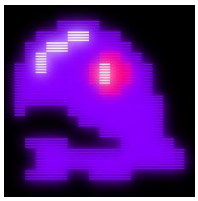
```
void Enemy::CollisionBubble(Bubble* bubble)
{
	captured = true;
	connect_bubble = bubble;
}
```

Figure 16-Function Called When Enemy Collides with a Bubble

To distinguish between different types of enemies, the enemy class includes enumerated enemy types (Figure 17) . This is later used by the enemy manager class, which needs to manage each type of enemy differently.

```
enum ENEMY_TYPE
{
	ZEN_CHAN,
	MONSTA,
	MAITA
};
```

Figure 17-Enumerated Enemy Types

ENEMY TYPES		
Zen Chan	Monsta	Maita
		
Zen Chan is the easiest enemy since it can only move horizontally. Its update function changes its direction if its x-axis velocity is zero (Figure 18). This allows Zen Chan to change direction every time it hits a wall or any other object.	Monsta uses similar techniques as Zen Chan but in both x and y-axis. Therefore, its update function changes its x-axis direction if its x-axis velocity is zero and the y-axis direction if its y-axis velocity is zero (Figure 19). This means that Monsta moves diagonally through the level creating a harder challenge for the player.	The most difficult enemy implemented is Maita. It continuously follows the player, with higher velocities the further away the player is. The update function gets the position of the player and calculates the direction Maita should travel. Since the direction is not normalised, it has a greater value the more the distance between the two positions (Figure 20). That value is later applied to the enemies velocity.

```
if (int(body->GetLinearVelocity().x) == 0)
{
	velocity = -velocity;
}
```

Figure 18-Zen Chan's Movement

```
if (int(body->GetLinearVelocity().x) == 0)
{
	velocity.x = -velocity.x;
}

if (int(body->GetLinearVelocity().y) == 0)
{
	velocity.y = -velocity.y;
}
```

Figure 19-Monsta's Movement

```
direction.x = (player_position.x - body->GetPosition().x);
direction.y = (player_position.y - body->GetPosition().y);

body->SetLinearVelocity({ direction.x * dt * velocity,direction.y * dt * velocity )};
```

Figure 20- Maita's Movement

For the game to work as intended, a non-fixed number of enemies was needed. In order to accomplish this, the Enemy Manager class was developed. This class is essential for creating and managing enemies. When Initialised, it receives spawning coordinates from the Level class, which are saved in an array.
Thereafter, when the "CreateEnemies" function is called, the manager picks a random spawning position using the previously mentioned array and a random number generator. It also uses a random number to create a random type of enemy. The number of enemies created depends on the current wave, received by the level as a parameter. Once the enemies are created, they are stored in a vector of Enemy pointers.
When the Enemy Manager's update or render function is called, the enemy vector is used to iterate through all enemy objects and update or render them respectively.
Lastly, when an enemy is killed, the "FindAndDestroy" function finds a specific enemy and uses the "DestroyEnemy" function (Figure 21) to delete the enemy's body from the physics world and the enemy's mesh.

```
void EnemyManager::DestroyEnemy(Enemy* enemy, b2World* world)
{
	//Destroy Body and Mesh
	world->DestroyBody(enemy->body);
	enemy->set_mesh(NULL);
	delete enemy;
	enemy = NULL;

}
```

Figure 21-Function used for deleting an enemy

Bubbles:

The Bubble game object is required to have a sphere mesh and a circle physics body. It should also start with a high initial velocity, not affected by gravity and able to capture any enemy it touches. Then it must decelerate at a constant rate reaching a low velocity, becoming unable to capture any enemies and starting to float.
The "Bubble" class (which inherits the "GameObject" class) has an initialisation function which creates the mesh and physics body and sets the initial velocity to a high value.
The update function of that class decelerates the bubble if its velocity is high. When the velocity is decreased to a low amount the bubble is disabled and no longer able to capture enemies. A gravitational force of negative value is also applied to the disabled bubble to make it float. In order to prevent the bubble from floating out of the level an if statement (Figure 22) checks whether its y-axis position is more than it should and sets its gravity to a positive value.

```
//Dont Allow Bubble to leave level
if (body->GetPosition().y > maxY-2)
{
	body->SetGravityScale(1);
}
else
{
	body->SetGravityScale(-1);
}
```

Figure 22-Code For Preventing the Bubble of Escaping the Level

For managing all bubbles in the game, a "BubbleManager" class was build. This class has all necessary function for creating, updating, rendering, and deleting bubbles. It also has an array of size 7 which can store up to 7 bubbles. This means that the player can't have more than 7 bubbles in the game at the same time.

When creating a new bubble, the manager checks whether the array of bubbles is full, if it is not then a new bubble is created next to the player with a high initial speed and a direction depending on where the player is facing (Figure 23).

```
void BubbleManager::CreateNewBubble(PrimitiveBuilder*
primitive_builder_, b2World* world, b2Vec2 position, float
player_sizeX, bool direction)
{
	for (int i = 0; i < 7; i++)
	{
		//If There are less than 7 bubbles
		if (bubbles[i] == NULL)
		{
			//Create New Bubble
			bubbles[i] = new Bubble;
			bubbles[i]->Init(primitive_builder_,
world, position, player_sizeX, direction, i);
			break;
		}
	}
}
```

Figure 23-The Bubble Manager Creating a new Bubble

For updating, the manager iterates through the array and if a cell points to a bubble object, then that object is updated.

Furthermore, when rendering the bubbles, the manager changes the override material of the 3D renderer to a half transparent green. Then a loop iterates through all bubbles of the array and renders them.

The "CollisionPop" function is called when the player collides with a bubble. This function deletes a given bubble's body and its mesh. This creates a place in the array where another bubble can be created later.

Fruits:

Fruits are simple but crucial game object types. Fruits are dropped whenever an enemy is killed and when collected offer the player some score points. Currently only one fruit is implemented in the game but the way the “Fruit” class is created, allows to easily add more fruits in the future. The Fruit class inherits from the “GameObject” class and includes, two functions: “Init” and “Update. The Initialisation function of the Fruit class creates a dynamic square physics body for the fruit at a given position, the bodies size is gathered by the size of the mesh. The Update function does nothing other than rotating the fruit, making sure it doesn’t fall of the level and building the transformation matrix for the fruits mesh (Figure 24).

```
void Fruit::Update(float dt, float maxY, float minY)
{
	//Spin Fruit
	rotationY+= dt*2;

	//If Fall Loop
	LoopAround(maxY,minY);

	BuildTransformationMatrix();
}
```

Figure 24-Fruit Update Function

Since a lot of fruit objects can be in the game at the same time, a “FruitManager” class was created to initialise, update, render and delete these objects. Whenever an enemy is killed, the “CreateFruit” class is executed, creating a new fruit at the popped bubbles position and pushing a pointer to it on the “fruits” vector (Figure 25). That vector is later iterated through to update and render all fruits. Each fruit also has a unique id integer value given to it when initialised, which is used for finding and deleting specific fruits using the “Delete Fruit” function (Figure 26).

```
void FruitManager::CreateFruit(PrimitiveBuilder*
primitive_builder_, b2World* world, gef::Vector2
position)
{
	//Create new Fruit
	Fruit* newFruit = new Fruit;
	//Set Mesh
	newFruit->set_mesh(fruit_mesh.mesh());
	//Set ID
	newFruit->personal_id = current_id;
	current_id++;
	//Init
	newFruit->Init(primitive_builder_, world,
position);
	newFruit->set_fruit_type(CHERRY);
	fruits.push_back(newFruit);
}
```

Figure 25- Creating a new fruit object

```
void FruitManager::DeleteFruit(int id, b2World* world)
{
	//Search for fruit
	for (int i = 0; i < fruits.size(); i++)
	{
		if (!(fruits[i] == NULL))
		{
			if (fruits[i]->personal_id == id)
			{
				//Delete fruit
				//Body
				world->DestroyBody(fruits[i]->body);
				//Mesh
				fruits[i]->set_mesh(NULL);
				delete fruits[i];
				fruits[i] = NULL;
			}
		}
	}
}
```

Figure 26-Deleting Fruit

Level Creation:

The "Level" class is necessary for building the game's level. Firstly, a vector of integers is created in the class's Initialisation function, filled with 0s or 1s. All ones in the vector represent tile blocks and all zeros represent empty space (Figure 27).

```
std::vector<int> map =
{

	1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,
	1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,
	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
	1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,
	1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,
	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
	1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,
	1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1

};
```

Figure 27-The Map Vector

Next, the “LevelBuilder” function is called which iterates through the “map” vector. Whenever a value of 1 is detected, this function calculates the corresponding position of the cell in the vector, to the world, and creates a static Game Object in that position. After created pointers to all the game objects are pushed in a vector called “platforms” (Figure 28).

```
for (int i = 0; i < map.size(); i++)
{
	//If is 1 Then Create Block
	if (map[i] == 1)
	{
		//Set POSITION
		gef::Vector2 pos = { 0,0 };
		pos.x = (i % 43)- 21;
		pos.y = -(( i- (i % 43) )/43 - 13);

		//Create Game Object
		GameObject* gameObj = new GameObject;

		//Init
		gameObj->Init(primitive_builder_, world, pos, { 0.5,0.5,1 });
		gameObj->body->SetAwake(false);
		gameObj->body->SetSleepingAllowed(true);
		platforms.push_back(gameObj);
	}
}
```

Figure 28-The Level Builder Function

Then, in the Render function a material is created using the tile texture and set as the override material of the 3D renderer. The “platforms” vector is later iterated through and all game objects are rendered.

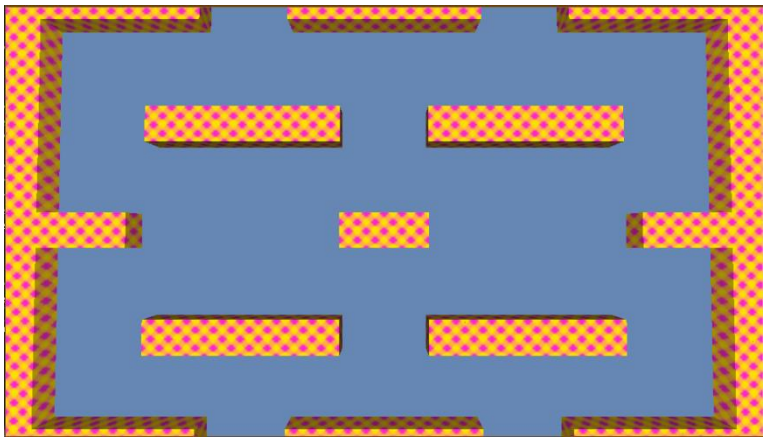


Figure 29-The Level

Models and Animation:

Model Loading:

All 3D models used in the game were created using a modelling software called "MagicaVoxel" (Ephtracy, n.d.). These models were exported as ".obj" files and converted to scene files using the model loader tool.

In the generic "State" class there are three important functions responsible for loading 3D Models in the game. The "LoadSceneAssets" function takes in the platform and a file name, creates, and returns a pointer to a scene (Figure 30).

```
if (scene->ReadSceneFromFile(platform, filename))
{
	scene->CreateMaterials(platform);
	scene->CreateMeshes(platform);
}
```

Figure 30-Loading Scene Assets from File

The "GetMeshFromSceneAssets" function takes a pointer to a scene as a parameter, extracts the mesh from that scene asset, and returns a pointer to it (Figure 31).

```
gef::Mesh* State::GetMeshFromSceneAssets(gef::Scene* scene)
{
	gef::Mesh* mesh = NULL;

	// if the scene data contains at least one mesh
	// return the first mesh
	if (scene && scene->meshes.size() > 0)
		mesh = scene->meshes.front();

	return mesh;
}
```

Figure 31-Extracting a Mesh from a Scene Asset

These two functions are used together in the "LoadAsset" function (Figure 32). This function just receives a filename and returns a pointer to a Mesh.

```
gef::Mesh* State::LoadAsset(const char* asset_filename)
{
	for (int i = 0; i < 3; i++)
	{
		gef::Scene* scene_assets_ = LoadSceneAssets(*platform_,
asset_filename);
		if (scene_assets_)
		{
			return GetMeshFromSceneAssets(scene_assets_);
		}
	}
}
```

Figure 32-Code for Loading Assets

Animation:

The “AnimatedMesh” class, which inherits from the Mesh Instance class, was created in order to implement stop-motion animation to the game. Using the “”AddFrame” function (Figure 33), a Mesh Instance pointer is created using a given mesh and saved in the “animation_frames” vector.

```
void AnimatedMesh::AddFrame(gef::Mesh* new_frame)
{
	gef::MeshInstance* new_Mesh = new gef::MeshInstance;
	new_Mesh->set_mesh(new_frame);
	animation_frames.push_back(new_Mesh);
}
```

Figure 33 - Function for Adding Frames to the Animation

The Update function is responsible for changing the animation frames and resetting the animation once all frames were displayed (Figure 34).

```
if (int(current_frame) == (animation_frames.size()))
{
	current_frame = 0;
}
else
{
	//Continue with the next frame
	set_mesh(animation_frames[int(current_frame)]->mesh());
	current_frame += dt * speed;
}
```

Figure 34-Updating the current frame

A little rotation animation is also created by the Update class whenever the direction changes. This is achieved by slowly increasing or decreasing the y-axis rotation until the mesh is completely rotated (Figure 35).

```
if (isLookingRight && rotationY > 0)
{
		rotationY -= speed*3*dt;
}
else
{
		if (!isLookingRight && rotationY < M_PI)
		{
			rotationY += dt*speed * 3;
		}
		else
		{
			if (rotationY < 0)
			{
				rotationY = 0;
			}
			else
			{
				if (rotationY > M_PI)
				{
					rotationY = M_PI;
				}
			}
		}
}
```

Figure 35- Rotation Animation

Physics and Collision Detection:

In the game, the Level state heavily uses collision detection and resolution. This is done in the "UpdateSimulation" function, where all contacts are received from the physics world and iterated through, using a for loop, resolving one by one. When a contact is detected two body pointers are created and each one is assigned one of the colliding bodies. Using "reinterpret_cast" those two bodies can be treated as game objects (Figure 36).

```
gameObjectA = reinterpret_cast<GameObject*>(bodyA->GetUserData().pointer);
gameObjectB = reinterpret_cast<GameObject*>(bodyB->GetUserData().pointer);
```

Figure 36-Reinterpereting the Bodies into Game Objects

From these pointers, the type of the game object can be gathered. Therefore, using if statements (Figure 37) collisions between specific objects can be resolved.

```
if (gameObjectA->type() == FRUIT && gameObjectB->type() == PLAYER)
{
	Player* playerObj = reinterpret_cast<Player*>(bodyB->GetUserData().pointer);
	Fruit* fruitObj = reinterpret_cast<Fruit*>(bodyA->GetUserData().pointer);

	//increase score and delete fruit
	playerObj->IncreaseScore(fruitObj->fruit_type());
	fruit_manager.DeleteFruit(fruitObj->personal_id, world_);
	break;
}
```

Figure 37 - Example of Collision Response

All collisions and resolutions include:

- Enemy with Bubble – Capture Enemy in Bubble.
- Player with Enemy – Player's health decreases.
- Player with Fruit – Player's score increases.
- Player with Bubble – Bubble is Popped.

Ghost Collisions and Resolution:

Ghost Collisions were a big problem faced while developing this game. Since the level was made by a lot of square bodies, the square players body continuously colliding with them created a lot of ghost collisions. This was resolved by changing the player's body shape to a circle. This decreased drastically the number of ghost collisions happening but not completely. Enemies (mostly Zen Chans) also created some ghost collisions since they have a square body. This

problem was not fixed but used as a game mechanic, since every time a ghost collision happened the enemy changed direction, making them less predictable, creating a fun challenge for the player.

Controls:

To offer more flexibility, both controller and keyboard inputs are acceptable for playing the game. In all Game States, controls are handled in the Update class. A pointer to both the input manager and controller manager are passed to each state when initialised. Using those pointers and an if statement (Figure 38), allows the user to control the game.

```
if (input_manager_->keyboard()->IsKeyDown(gef::Keyboard::KC_RIGHT)
|| (controller_->left_stick_x_axis() >0))
{
	player.walk(true, dt);
}
```

Figure 38 - Code for Controlling the Player

Audio:

The music for this game was created using “Beepbox” (John, n.d.) and the sounds were created using “Jsfxr” (Eric, 2011). All sounds were exported in “.wav” format and loaded into the game’s audio manager. Each music is loaded and played at the beginning of each state(Figure 39).

```
void LevelState::OnEnter()
{
	//Play Level Music
	audio_manager_->LoadMusic("sfx/level_music.wav", *platform_);
	audio_manager_->PlayMusic();
}
```

Figure 39 - Loading and Playing the Level Music

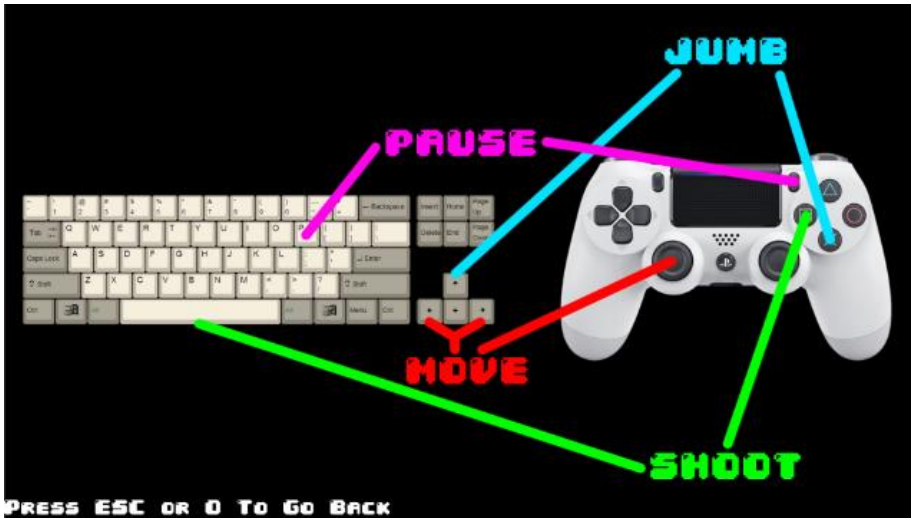
Using the audio manager, each state can then play sounds depending on the condition needed (eg. Collision, choice change, movement) (Figure 40). The audio manager is also used when changing the volume in the options menu.

```
if (input_manager_->keyboard()->IsKeyPressed(gef::Keyboard::KC_UP)
|| (controller_->buttons_pressed() & gef_SONY_CTRL_CROSS))
{
	if (player.jumb())
	{
		audio_manager_->PlaySample(0, false);
	}
}
```

Figure 40 - Playing the Jumping Sample when Player Jumps

User Guide:

The user can go through the menu and options using the arrow keys or the left controller stick. To choose something, Space or X can be pressed. Once in the game the player can move, jump, and shoot with the following controls:



The aim of the game is to get as higher score as possible. To gain any score the player has to shoot at the enemies and capture them in bubbles. Once the enemy is captured the bubble can be popped by the player to get cherries, which when collected increase the score. The player can have up to 7 bubbles on screen at the same time, so popping bubbles allows you to create even more bubbles.

Data Oriented Design:

In contrast to the Object-Oriented Design, Data-Oriented Design focuses on how data is stored in the computer's memory and how efficiently it can be fetched and processed (Mines, 2018). Because the Object-Oriented Design was used while creating the game, a lot of cache misses happen which result to the application not processing data as efficient as possible. A way to improve this problem is to mix some Data-Oriented Design elements into the applications code. This can be used on all manager classes.

Having a bubble manager class with pointers to objects of type Bubble is very inefficient since every time a bubble needs to be processed, the CPU has to wait for the bubble to be found in

RAM. This can be improved by having an array of bubble items instead. This would be a very simple improvement since when iterating through all bubbles to update or render them, the Bubble data would be already loaded to the cache. This would still have some faults since some bubbles would be active, but some wouldn't. Therefore, the CPU would have to jump to a different part of the cache every time it would find a deactivated bubble. To solve this issue, bubbles could be sorted; for example, keeping active bubbles to the front and non-active bubbles to the back of the array. This would lower the cache misses even more since all bubbles that can be updated and rendered, are at consecutive memory locations. The same design can be applied to the fruit manager, enemy manager, and level class.

This design can be improved even more by limiting the use of game objects. Game Objects take a lot of space in memory therefore preventing more information to be loaded into the cache. The level class can be completely reconstructed by having 2 arrays, one for holding all bodies and one for holding all meshes of all the platforms used to create the level (Figure 41). Then the update function could use both of those arrays to update the platforms and the render function could use only the meshes array to render everything. This could be more efficient because the cache can hold a higher number of meshes than game objects since they take up less memory.

```
std::vector<gef::MeshInstance> tile_meshes;
std::vector<b2Body> tile_bodies;
```

Figure 41-The two arrays holding information for the level

Reflection:

During this semester, I managed to create a 2.5D game. The game is both fun and challenging which makes it addictive to the user. A lot of key techniques, most of which mentioned above, were successfully implemented creating a polished final outcome. Although difficult, I found the development process, quite enjoyable. Many hours were spent working on solving problems and overcoming obstacles but looking at the finished application makes me believe that it was all worth it. Of course, there is still a lot of room for improvement. More fruits and enemies could have been added as well as a level selection feature. If I would do this again, I would spend much more time working on the application which would allow me to add more features and cleaner code. I would also use a mixture of both Object-Oriented and Data-Oriented Designs since the Data-Oriented Design offers substantial performance improvements. To conclude, creating this game was a pleasant challenge which helped me improve my problem solving and programming skills.

References

Ephtracy, n.d. *MagicaVoxel.* s.l.:s.n.

Eric, F., 2011. *Jsfxr.* [Online]
Available at:
https://sfxr.me/#11111GmbDSffi5R4hJYTSdVzvspRKLNYekd8ukLxxKa1MjgPK3s6E6iywexwp4o8SAkRnW6muYbZ4xa8hhpd7VVRkQoXLurnUYzixMU9RYi7ZAxhkPHxyzDM
[Accessed 2 May 2021].

John, N., n.d. *BeepBox.* [Online]
Available at:
https://www.beepbox.co/#https://www.beepbox.co/#8n31s0k0l00e03t2mm0a7g0fj07i0r1o3210T3v1L4uf3q1d5f8y2z7C0SU0M51pr2qiqqrrrT5v1L4ud3q1d2f7y3z1C0c0h0HVxh90000000000T1v1L4u99q3d7f6y3z1C0c0AcF8BcV9Q4200P6789E0000T2v1L4u15q0d1f8y0z1C2w0b4h400000000h4g000000014h
[Accessed 2 May 2021].

Knight, S., n.d. [Art].

Mines, J., 2018. *Data-Oriented vs Object-Oriented Design.* [Online]
Available at: https://medium.com/@jonathanmines/data-oriented-vs-object-oriented-design-50ef35a99056
[Accessed 14 May 2021].