**GDV4002 – Applied Maths for Games**

**Workshop Handbook ver0.8**

Contents

[1. Overview 2](#_Toc215497317)

[2. Adding Objects 6](#_Toc215497318)

[3. Setting Object Properties 10](#_Toc215497319)

[4. Updating Objects in the Game Loop 14](#_Toc215497320)

[5. Animating Multiple Objects 20](#_Toc215497321)

[6. Getting User Input – Keyboard input 28](#_Toc215497322)

[7. Using Object-Oriented Techniques for in-game objects 36](#_Toc215497323)

[7.1. Adding the Player Class 38](#_Toc215497324)

[7.2. Setting up the Player header (.h) file 40](#_Toc215497325)

[7.3. Creating the Player implementation in the cpp file 42](#_Toc215497326)

[7.4. Creating a Player instance in the game scene 44](#_Toc215497327)

[7.5. Creating an Enemy class 45](#_Toc215497328)

[7.6. Summary 47](#_Toc215497329)

[8. Using physics to control a character 48](#_Toc215497330)

[8.1. Game physics – basic attributes 49](#_Toc215497331)

[8.2. Adding physics to the Player class 50](#_Toc215497332)

[8.3. Adding gravity 54](#_Toc215497333)

[8.4. Impulse Forces 55](#_Toc215497334)

[9. Creating and destroying in-game objects: Particle System example 57](#_Toc215497335)

[9.1. Anatomy of a Particle System 57](#_Toc215497336)

[9.2. Creating a Particle Class 60](#_Toc215497337)

[9.3. Creating an Emitter Class 63](#_Toc215497338)

[9.3.1. Emitter Definition 63](#_Toc215497339)

[9.3.2. Emitter Implementation 66](#_Toc215497340)

[9.4. Using an Emitter 71](#_Toc215497341)

[9.5. Deleting Particles 72](#_Toc215497342)

[10. Controlling the view 75](#_Toc215497343)

[10.1. Setting the viewplane size at window creation time 76](#_Toc215497344)

[10.2. Setting the viewplane after the window has been created 77](#_Toc215497345)

# Overview

In order to focus on the mathematical ideas being introduced in this module, writing code to handle any game rendering and user input can get in the way! This is the purpose of the gdv4002-base1 project (and subsequent versions!). It provides a base-line engine that you can use to setup windows, run a game loop and render in-game objects.

The project can be forked and cloned from <https://github.com/pna21842/gdv4002-base1.git>

A screenshot of a computer

AI-generated content may be incorrect.

When you download the project and open the visual studio solution, you’ll see a folder named ‘Engine’ and under the “Source Files” folder you’ll see main.cpp. The project has been designed so that you do not need to edit any code in the Engine source files. **All the work you do will be in main.cpp**. As you progress and get introduced to Object Oriented concepts, you might want to add your own source files for you own classes, but we will come to this later.

*Figure 1.1 Project solution*

Let’s have a look at the code in main.cpp…

A computer screen shot of a program code

AI-generated content may be incorrect.

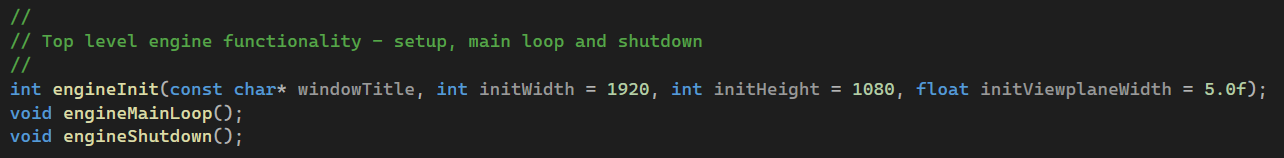
*Figure 1.2 Initial main.cpp file*

The only header it includes is “Engine.h”. This contains the main interface for the engine, that is, the functions you’ll use to setup objects etc.

The main function is setup to do 4 things:

* Initialise the engine and create the window. Here we can specify the title of the window and its initial width and height.
* Setup scene objects - this is just shown as a comment – we’ll add to this later.
* Once the game scene objects have been setup we enter the main game loop. The game stays in game loop the user quits.
* The final step is to shut down the engine – this deletes resources and prints final timing data, before returning 0 for success!

Let’s look at some of the things you can do with the engine itself. If we open Engine.h we can see a range of functions in different groups. The first group refers to the top-level functions of the engine…



*Figure 1.3 Top-level engine functions*

These are called already in the initial code shown in main.cpp, so we don’t really need to call these ourselves.

The next group of functions are related to *event registration*…

A screen shot of a computer code

AI-generated content may be incorrect.

*Figure 1.4 Event registration functions*

The engine allows us to provide our own function to handle keyboard input, drawing and per-frame updates of the game objects. We’ll likely not need to change the render (draw) function, but we will look at the update function and keyboard input later.

The main group of functions come under the “Update / Query engine state” section. These functions allow us to check on the engine configuration, find objects we want to update, add new objects, delete objects and change basic rendering properties.

A screenshot of a computer program

AI-generated content may be incorrect.

*Figure 1.5 Query / Update functions*

The final group of functions are for testing purposes – they can be used to print information about the game objects currently being managed by the engine.

A screen shot of a computer code

AI-generated content may be incorrect.

*Figure 1.6 Debug functions*

The engine itself is based on a number of existing libraries. These are summarised below:

**OpenGL**: This is an industry standard graphics API and it underpins the drawing (rendering) and texture / sprite handling. ([OpenGL - The Industry's Foundation for High Performance Graphics](https://www.khronos.org/opengl/)). Note: OpenGL has largely been superseded by the newer [Vulkan API](https://www.vulkan.org/).

**GLEW**: This stands for the Open**GL E**xtension **W**rangler library. It provides a single interface to setup and access all the OpenGL functionality up to and including the latest release (4.6). ([GLEW: The OpenGL Extension Wrangler Library](https://glew.sourceforge.net/))

**GLFW**: The GLFW library provides a cross-platform API to allow you to easily setup an OpenGL rendering environment, windows to display output as well as handling input and output. Without GLFW and similar libraries, you’d need to use the native platform APIs to manage such aspects which can be cumbersome. ([An OpenGL library | GLFW](https://www.glfw.org/))

**FreeImage**: An open source project to enable you to work with different image formats including png, jpeg, bmp, tiff etc. The game engine uses FreeImage to load image data for sprites. ([The FreeImage Project](https://freeimage.sourceforge.io/)).

**GLM**: The OpenGL Maths library provides numerous types and constants to enable you to work with vectors and matrices directly in C++ ([OpenGL Mathematics](https://glm.g-truc.net/0.9.9/index.html)).

Let’s start by creating new objects in the game scene…

# Adding Objects

The engine stores a collection of game objects that have the type GameObject2D and the engine provides functions to add objects to the scene. The main function to do this is called addObject. There are 2 versions of the addObject function. The first version allows us to specify the properties of the object in directly; the other allows us to add an already existing object. Let’s look at creating an object using the first version – by specifying the properties directly as parameters. Back in main.cpp, look for the comment in the main function…

A screen shot of a computer

AI-generated content may be incorrect.

We’ll put all our setup code after this!

If we look at the addObject function in the engine.h file, we see it takes a number of parameters…

GameObject2D\* addObject(

const char\* name,

glm::vec2 initPosition = glm::vec2(0.0f, 0.0f),

float initOrientation = 0.0f,

glm::vec2 initSize = glm::vec2(1.0f, 1.0f),

const char\* texturePath = nullptr,

TextureProperties texProperties = TextureProperties())

At the very least we need to specify a name for your new game object. You can get game objects out of the engine later using the name string to access them. Let’s create a new object by just specifying its name… (remember, put this after the “Setup game scene objects here” comment).

A screen shot of a computer

AI-generated content may be incorrect.

addObject(“player”);

If we run the code, we get an object in its default position (x=0, y=0), size and orientation. There’s no sprite image attached so it’s just a white square – not very interesting!

A screenshot of a computer

AI-generated content may be incorrect.The next parameter we can add it the position vector. We’re using the glm library so we can tie our position coordinate into a single parameter of type vec2 (2D vector) rather than have x and y specified separately. Let’s set a new initial position…

addObject("player", **glm::vec2(1.0f, 1.0f)**);

When we run this the player object appears at its new position. We’ll look at the coordinate range in a later section.

A screenshot of a computer

AI-generated content may be incorrect.Next, let’s add in the orientation, or angle of rotation. As discussed in lectures, we typically specify angles in radians. So, if we want our player to be rotated by 45 degrees, we need to convert this by multiplying 45 by pi/180. Let’s look at adding the orientation in…

addObject("player", glm::vec2(1.0f, 1.0f), **45.0f \* pi / 180.0f**);

The glm library actually comes with a function to do the radians conversion for us (glm::radians), so the above is the same as…

addObject("player", glm::vec2(1.0f, 1.0f), **glm::radians(45.0f)**);

A screenshot of a computer

AI-generated content may be incorrect.The next parameter we can add is the size of the object. Size is specified as a width (size on the x axis) and height (size on the y axis). This defaults to 1.0 in both directions. Let’s change this so we half the width but leave the height the same. Again, we can use a glm::vec2 type to combine the width and height into a single parameter…

addObject("player", glm::vec2(1.0f, 1.0f), glm::radians(45.0f), **glm::vec2(0.5f, 1.0f)**);

When run we can see that, although rotated, the player object is now a rectangular shape.

The last two parameters allow us to add a texture image. A texture is a term used in graphics programming to refer to an image mapped into an object. We can also refer to this as a sprite as well. The first of these two parameter specifies the file path to the image we want to use. There are already some examples in the Resources\Textures folder A green and brown rectangle

AI-generated content may be incorrect.(this can be loaded in the main project sub-folder). Let’s add the mcblock example to our player object…

addObject("player", glm::vec2(1.0f, 1.0f), glm::radians(45.0f), glm::vec2(0.5f, 1.0f), **"Resources\\Textures\\mcblock01.png"**);

**Quick note**: when specifying paths, we use a double ‘\’ character to separate elements in the path. Remember, special characters in C++ strings are formed with a \ such as \n for newline or \t for tab. \\ tells C++ we just want a ‘\’ character.

Also, we don’t need to specify the entire path i.e. C:\......\Resources\...etc. This would be an absolute path from the root of the host drive. By leaving out C:\ the path becomes a relative path. But relative to what? When we run our project from within Visual Studio, it searches for the path starting from the location of the project folder. This is why Resources is located next to the project files! When we build an exe and run the exe directly from Windows or the command line, it searches for the path starting from the location of the exe.

As an exercise, try adding in one of your own images to see what the results would be. You can store your image anywhere, but when working on assets (textures, sprites, audio files, 3D models etc) it’s good to be organised and be consistent! So, it’s recommended you store your own images in the Textures folder too.

The last parameter to the version of addObject we’ve been using specifies the texture properties. You can generally ignore this as we’re more interested in using the engine for maths not graphics. For most cases, the default is fine, but in some cases, as is the case of the MineCraft example above, we may want to change the properties to show a more pixellated, retro look. To achieve this effect we can set the last parameter to TextureProperties::NearestFilterTexture() as shown below…

A pixelated rectangular object with green and brown squares

AI-generated content may be incorrect.

addObject("player", glm::vec2(1.0f, 1.0f), glm::radians(45.0f), glm::vec2(0.5f, 1.0f), "Resources\\Textures\\mcblock01.png", **TextureProperties::NearestFilterTexture()**);

As can be seen, the result looks more pixellated compared to the image shown earlier.

This walk-through summarises the use of addObject to specify the individual properties of our game object. In the next section we’ll look at modifying properties of existing game objects.

# Setting Object Properties

In this section we’ll look at obtaining game objects stored in the engine and setting or changing their properties. This section will focus on doing this in the setup code in the main function, but the same methods can be used to get and modify objects in the update function which we’ll look at in a later section.

Let’s first create 2 objects as follows…

A screenshot of a computer

AI-generated content may be incorrect.

addObject("player1");

addObject("player2");

If we run this, we can see what appears to be a single object, but both game objects are being drawn, it’s just that both use the default position of (0, 0) so they’re overlapping.

Let’s access both players and change their positions. The demo engine stores game objects of type GameObject2D – we can query a game object with a specific name as follows…

GameObject2D\* player1Object = getObject("player1");

Note the ‘\*’ at the end of the GameObject2D type. This tells us it’s a pointer. However, don’t worry about memory management – we’re just using pointers and not changing them in any way.

Once we have player1Object, let’s check it’s not null – a null pointer is a way of indicating the engine couldn’t find an object with the name “player1”.

if (player1Object != nullptr) {

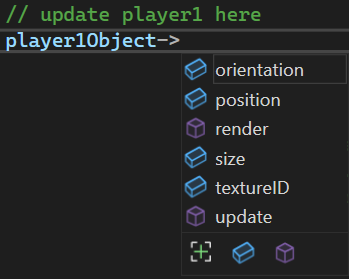
// update player1 here

}

Once we know player1Object is okay, we can go ahead and change the properties. If you type the following just below the “update player1 here” comment…

player1Object->

…IntelliSense should kick in and show you what you can use / change within the game object…



The items with the blue icon are the attributes / properties we can change (the items with the purple icon are the functions on the object – we don’t call these directly).

A screenshot of a computer

AI-generated content may be incorrect.

Let’s set the position of player 1. We’ll use the glm::vec2 type as before…

player1Object->position = glm::vec2(-1.0f, 1.0f);

When we run this, we can see the player1 object has moved to the left side of the window.

A screenshot of a computer

AI-generated content may be incorrect.Let’s repeat what we did for player1 for player2. After the if() block where we checked player1Object was not null, add the following code to get the “player2” object and set it’s position…

GameObject2D\* player2Object = getObject("player2");

if (player2Object != nullptr) {

player2Object->position = glm::vec2(1.5f, 1.0f);

}

Running this now shows the player2 object moved to the right of the window.

A screenshot of a computer

AI-generated content may be incorrect.While we have the player objects, we can also set other properties. To set the orientation add the following to the respective if{} blocks setup above…

For the player1 object add…

player1Object->orientation = glm::radians(-30.0f);

For the player2 object add…

player2Object->orientation = glm::radians(45.0f);

We can now see both objects rotated by -30 degrees and 45 degrees respectively.

As an exercise have a go at setting the size property. Remember, the size property takes a glm::vec2 to combine the width and height values together.

At the moment, our game objects have no sprites assigned. Let’s change this by setting the textureID property. The game engine has a loadTexture function that takes a path to the texture file and optionally, a texture properties object. We’ll leave the texture properties parameter for now. The loadTexture function sets up an image and returns the image ID code which we’ll store in our textureID property. The type of the image ID code is GLuint – this is just an unsigned integer.

**Note**: Textures (sprites) are managed by OpenGL. GLuint is just an alias defined as part of OpenGL’s headers to represent a standard unsigned int.

A screenshot of a computer

AI-generated content may be incorrect.Let’s have load the bumblebee demo texture in the Resources\Textures folder and assign this to the player1 object. Inside the player1Object if{} block add…

player1Object->textureID = loadTexture("Resources\\Textures\\bumblebee.png");

When we run this, we can see that the player1 object now has the bee image attached to it!

**Exercise**: As an exercise, use the same approach to add a texture to the player2 object. Use one of the example images or add your own.

**Remember**: You can store your image anywhere, but when working on assets (textures, sprites, audio files, 3D models etc) it’s good to be organised and be consistent! So, it’s recommended you store your own images in the Textures folder too.

# Updating Objects in the Game Loop

In order for objects to move and animate in a game, we need to change their properties such as position, orientation etc. on a *regular basis*. This so-called regular basis is *every* *frame*. If you’re unsure what is meant by frame, let’s look at what happens when we enter the game loop.

In the main function we put all our necessary setup / initialisation code. After this we call engineMainLoop. This is where our game lives until we quit. Let’s have a look inside this function to see what’s going on…

**while (!glfwWindowShouldClose(window**)) {

// 1. Update game clock and get time elapsed since last time round the loop!

gameClock->tick();

double tDelta = gameClock->gameTimeDelta();

// 2. Update game environment (animations etc)

if (overrideUpdateFn)

overrideUpdateFn(window, tDelta);

else

defaultUpdateScene(tDelta);

// 3.1. Render current frame

defaultRenderScene();

// 3.2. Display newly rendered frame

glfwSwapBuffers(window);

// 4. Poll events ie. user input (key presses, mouse events)

glfwPollEvents();

// (optional) Update window title to show current fps / spf

char timingString[256];

sprintf\_s(timingString, 256, "%s: Average fps: %.0f; Average spf: %f", windowTitleString.c\_str(), gameClock->averageFPS(), gameClock->averageSPF() / 1000.0f);

glfwSetWindowTitle(window, timingString);

**}**

Each time through the loop we update the game objects / scene and redraw everything. Each time through the game loop is called a frame, as it outputs a new frame of animation for the game. If the loop repeats quick enough then we output each frame, or image fast enough to look like a continuous animation. Many games aim to get through the loop quick enough that the loop repeats around 60 times per second (hence 60 frames per second, for fps).

To manage our own updates, we can use a feature called a callback function. This is where we can tell the engine to use our own function instead of its own built-in update function.

Let’s add our own update function into our code. Back in main.cpp, at the top of the file you’ll see a comment // function prototypes. Here we’ll put what we call the function signature (it’s return type, name and parameter list). Let’s add that now below the comment…

void myUpdate(GLFWwindow\* window, double tDelta);

This tells C++ that we’re going to define this function later. After the main function we’ll add the main function definition for myUpdate…

int main(void) {

// main function body

}

**void myUpdate(GLFWwindow\* window, double tDelta) {**

**}**

The function definition must match the prototype we included before. In addition, we can’t change the parameters for the function – the engine expects to pass the pointer to the main window and a double representing the time elapsed since the last frame.

Now that we’ve defined our own update function, let’s go back to out setup code in the main function and tell the engine to use our new function for updates.

After any object setup code and ***before*** we call engineMainLoop(), call setUpdateFunction to tell the engine to use our new myUpdate function instead of the default. We pass the name of our myUpdate function as the parameter…

setUpdateFunction(myUpdate);

If we run our game nothing happens because our update function is empty. Let’s change that! If you didn’t complete section 2 or 3 above to add game objects to the scene let’s do that now. If you haven’t setup any player objects add the following after the // Setup game scene objects here comment…

addObject("player1", glm::vec2(-1.5f, 1.0f), glm::radians(-30.0f), glm::vec2(1.0f, 1.0f), "Resources\\Textures\\bumblebee.png");

addObject("player2", glm::vec2(1.5f, 1.0f), glm::radians(45.0f), glm::vec2(1.0f, 1.0f), "Resources\\Textures\\bumblebee.png");

**Let’s have a go at animating player 1 so it rotates on the spot**. At it’s most fundamental any animation needs 3 things

* a property to change,
* a variable to represent that change,
* a function to apply the change to the property
* and a function to draw the result.

We already have two of there. The game object contains the orientation property to control how much the object is rotated. For this example, we’ll be changing this in the update function. In addition, the game engine already has the ability to render (draw) our game objects so we can see the result.

What we don’t have are the middle two items:

* a variable to represent that change,
* a function to apply the change to the property

Let’s add these in!

The variable to represent change should be specified as t*he amount of change per second*. So, if we expect our object to rotate at 45 degrees per second we’d store the value as such(converted to radians of course)…

float anglesPerSecond = glm::radians(45.0f);

If we wanted to move a player 2 units within the game world we’d store this as follows:

float playerVelocity = 2.0f;

For our example (rotating player1), we’ll set it so it rotates 90 degrees every second. In the myUpdate function we setup earlier let’s add this in…

void myUpdate(GLFWwindow\* window, double tDelta) {

**float player1RotationSpeed = glm::radians(90.0f);**

}

Next we want to update the orientation property for player1. To do this we first get the game object representing player1…

GameObject2D\* player1 = getObject("player1");

Then we add the rotation speed to player 1’s orientation. However, before we do this, we must scale (multiply) the rotation speed by the tDelta variable. tDelta represents the time elapsed since the previous update and must be multiplied with our speed variable. This is done as follows…

player1->orientation += player1RotationSpeed \* tDelta;

Running the game will now show player1 rotating on the spot!

**Discussion**: So, what’s the deal with tDelta? Why not just have a smaller constant speed to add to the orientation each frame? Let’s do a thought experiment to see what would happen. Let’s say you’re developing on a machine that refreshes its display at 60 Hz (60 times a second) and your game runs at a nice 60 frames per second to go with this. You calculate that 90 degrees divided by 60 is a neat 1.5 degrees to rotate per frame. So, you write your update code as…

player1->orientation = player1->orientation + glm::radians(1.5);

Running your code, you get a perfect rotation of 90 degrees every second. Now, you give a copy of your game to a friend (Kate) who’s running a much higher spec machine with 120hz refresh rate and as such the game runs at 120 frames per second. Remember, a frame is 1 trip through the game loop, which means your update function gets called 120 times in 1 second, not 60! Since you’ve hard-coded 1.5 degrees as the amount of change per frame, after 1 second we have 120 \* 1.5 = 180 degrees of rotation! Kate sees the rotation twice as fast as you do!

You also gave you game to another friend (Bob) who, in our thought experiment, has a slow machine that runs your game at 30 frames per second. This means your update function gets called 30 times in 1 second, which means the amount of rotation after 1 second is equal to 30 \* 1.5 = 45 degrees of rotation. Bob sees the rotation running slower!

Running the game on all 3 machines side-by-side would show the rotations are actually running at different speeds. This may impact gameplay – too slow for Bob and too fast for Kate.

Now, let’s go back to tDelta and add this back in. You change your code to the following…

player1->orientation = player1->orientation + glm::radians(90) \* tDelta;

Now, on your machine running at a constant 60fps, tDelta will be 1 second divided by 60fps (1/60), which is approximately equal to 0.016 seconds per frame. Bob’s machine runs at 30fps, so tDelta will be 1 / 30 = 0.0333 and Kate’s machine running at 120fps will have a tDelta of 1/120 = 0.008.

So, every frame your machine rotates player1 by 1.5 degrees as was the case above. However, Bob’s machine rotates the player by 0.0333 \* 90 degrees which is approximately 3 degrees. And Kate’s machine rotates the player by 0.008 \* 90 which is approx. 0.75 degrees.

So, Bob’s machine rotates by more degrees per frame but it does this less frequently (30 times a second). Kate’s rotates less per frame (0.75 degrees) but does this more frequently (120 times a second).

Over 1 second the total rotation of player1 on each machine becomes:

Your machine: 1.5 \*60 = 90 degrees over 1 second

Bob’s machine: 3 \* 30 = 90 degrees over 1 second

Kate’s machine: 0.75 \* 120 = 90 degrees over 1 second

So, by scaling the speed of rotation by the time elapsed each frame you can ensure the speed of rotation on each machine is consistent, so Bob and Kate have the same experience as you. But this isn’t just about rotation, it applied to any value that represents change.

# Animating Multiple Objects

In the previous section we animated a single object in our new update function. In this section we’ll have a go at setting up multiple objects. For this walkthrough let’s setup 3 enemy objects that are oscillating up and down on the screen. As part of this we’ll look at setting up object “groups” that we can retrieve and process together.

A pixel art of a green monster

AI-generated content may be incorrect.

First things first – let’s find a sprite image we can use for our bad guys. Feel free to create something yourself or search for an image online. Here’s something created with Paint’s Co-Creator (I’ll include this on Moodle)…

*Figure 5.1. Example enemy sprite*

Next, we’ll create our game objects for each of the enemies. For this example, let’s clear out the game objects we created before and add in the following to make a new scene. This creates a single “player” and 3 enemy objects. **Hint**: If you don’t want to lose your hard work from before then don’t worry, you can just comment out your old scene setup code. If you did delete it then thanks to git, it’ll still be there in an earlier commit!! In the main function (in your scene setup area) add the following…

addObject("player", glm::vec2(-1.5f, 0.0f), 0.0f, glm::vec2(0.5f, 0.5f), "Resources\\Textures\\player1\_ship.png");

addObject("enemy", glm::vec2(0.0f, 0.0f), 0.0f, glm::vec2(0.75f, 0.75f), "Resources\\Textures\\alien01.png");

addObject("enemy", glm::vec2(1.0f, 0.0f), 0.0f, glm::vec2(0.75f, 0.75f), "Resources\\Textures\\alien01.png");

addObject("enemy", glm::vec2(2.0f, 0.0f), 0.0f, glm::vec2(0.75f, 0.75f), "Resources\\Textures\\alien01.png");

When you run this you should see something like the following…

A screenshot of a computer screen

AI-generated content may be incorrect.

*Figure 5.2. Initial Scene*

An important point to note here is the *name* used for the enemy objects. Up until now we’ve used a unique name for each object, but the engine allows us to have groups of objects under the same name. Here we’re calling all our enemies “enemy”.

**Note**: Behind the scenes, the engine keeps track of this and creates a unique key for each member of the “enemy” group. Let’s use those debug functions outlined above to see what’s going on in the engine.

In your setup code, just before you call engineMainLoop, call…

listGameObjectKeys();

This will list something like the following (the order may be different for you)…

player

enemy

enemy2

enemy3

Notice that though we added 3 “enemy” objects, the engine has appended a number to each subsequent one added, giving each object in the scene a unique name after all!

In terms of object groups, the engine also keeps track of this too. After your call to listGameObjectKeys(), call the other debug function…

listObjectCounts();

This will produce the following…

Object key counts...

player has count 1

enemy has count 3

As you can see, there are two object types, or object groups being tracked – “player” has a count of 1 – our single player – but “enemy” has a count of 3.

**Note**: You don’t need to use this facility – you could give each enemy its own unique name. However, when we look at updating objects this feature can come in handy.

So, now we’ve setup our scene, let’s head over to the update function we created before and get our enemies animated! As before, if you’ve got code from the previous exercise or your previous experiments in the update function, delete or comment this out – we’re going to do some new animations here!

For our animations we want the enemies to oscillate up and down in the following manner. The enemy will move to the top of the screen, slow down, then start moving towards the bottom of the screen. When it gets to the bottom of the screen it will slow down again and start moving back to the top – repeating this pattern. Only the *y* coordinate (vertical position) will change – the enemies *x* position will not – the path shown here is just for illustration only!

*Figure 5.3. Enemy movement pattern*

How can we do this? There are many ways to accomplish this – some more efficient and easier to code than others. The first thing we need is the position of the enemy as this changes in the animation. No problem – we have this already in the game objects we created above. We also need information about where in the animation cycle (up, down, up again etc) we are!

If we look at the above diagram and the movement path, does this remind you of anything (**Hint**: checkout the trigonometric functions we looked at previously) This looks like a sine (sin) function! So, for a given enemy, why not store an “angle”, or more precisely a “phase” variable to say where in the animation we are. This “phase” value will move from 0 through π to 2π (and so on) since this is the parameter range (domain) for the sin function.

Hang on a sec, I’m not rotating anything, just moving up and down, why do I need an angle, or phase variable and the sin function? That’s true, and we’re not using this to do rotation, which is why “phase” is more appropriate than “angle” as a variable name. While we’re not doing rotation, we are interested in the shape of the sin function. Let’s check this out…

As the phase increases to π/2, the y coordinate increases to 1 (moves up)

As phase heads towards 2π, the y coordinate heads back to 0 (where we started)

A pixel art of a green monster

AI-generated content may be incorrect.A graph of a function

AI-generated content may be incorrect.

A pixel art of a green monster

AI-generated content may be incorrect.

A pixel art of a green monster

AI-generated content may be incorrect.

A pixel art of a green monster

AI-generated content may be incorrect.

At phase=0, sin(0)=0 so the enemy’s y position is 0

As the phase increases through π/2 towards 3π/2, the y coordinate decreases to -1 (moves down)

*Figure 5.4. Sin wave for enemy movement*

Now, sin (like cos) is a periodic function, so it repeats every 2π cycle…

A graph of a graph

AI-generated content may be incorrect.

+∞

-∞

*Figure 5.5. sin is a periodic function so repeats every* 2π

… so, as our phase variable increases to infinity, our sin function (and y coordinate) just oscillates between -1 and +1.

So, in summary, from the above we know we need a phase variable to represent where we are on the sin wave, and we’ll use sin(phase) to calculate the enemy’s *y* position.

But in our update function we’ll need to change the phase as well, otherwise it stays constant and our enemy stays at the same *y* coordinate. So along with the phase variable we’ll also define a velocity that says how many degrees the phase variable changes *per second*. For example, if we want our enemy to move at a speed so they go from the middle to the top of the screen in 1 second, we can see in the above sin wave diagram we need our phase variable to increase by 90 degrees over 1 second, so we’d set the velocity to glm::radians(90.0f).

Now, when we setup the scene above we had 3 enemies, so where going to need 3 phase variables – one per enemy – one 3 phase velocity variables. We could store these as phase1, phase2 and phase3 for example, but when working with multiple objects (collections of objects), we usually want to use some sort of array. Here we’ll create 2 arrays – one for the enemy phase positions and one for their corresponding velocities.

Just *above* your myUpdate function add the following arrays (since they’re not included in the body of any function they’re global variables)…

float enemyPhase[3] = { 0.0f, 0.0f, 0.0f };

float enemyPhaseVelocity[3] = {glm::radians(90.0f),

glm::radians(90.0f),

glm::radians(90.0f) };

Here we set all enemies to have the same phase and same phase velocity.

To update all the enemies, we could get them one at a time using the getObject function shown in section 4 above. However, the engine also provides a getObjectCollection function to get all objects that share the same key.

Let’s have a go at adding this in for just one enemy. We’ll setup our update function as follows to get the object collection:

void myUpdate(GLFWwindow\* window, double tDelta) {

GameObjectCollection enemies = getObjectCollection("enemy");

}

The returned GameObjectCollection ***enemies*** contains 2 items:

enemies.objectCount Number of returned objects

enemies.objectArray Array of game objects matching the key “enemy”

Let’s have a go at updating the first enemy in this array (the one at index 0). We can access the game object directly with…

enemies.objectArray[0]

…or we could assign this to a local variable as follows…

GameObject2D\* objectToProcess = enemies.objectArray[0];

Let’s go with the first approach for now. Since we’re updating the object at index 0, we use enemyPhase[0] to calculate it’s y position, and enemyPhaseVelocity[0] to change the enemyPhase[0] variable so we get our animation.

To set the first enemy’s position based on it’s phase we can say…

enemies.objectArray[0]->position.y = sinf(enemyPhase[0]); // assume phase stored in radians so no conversion needed

To update the phase value, we use the phaseVelocity. But since this is the change-per-second, we always multiply this by the time elapsed (tDelta)…

enemyPhase[0] += enemyPhaseVelocity[0] \* tDelta;

If we add these 2 lines to the update function so it looks like the following…

void myUpdate(GLFWwindow\* window, double tDelta) {

GameObjectCollection enemies = getObjectCollection("enemy");

enemies.objectArray[0]->position.y = sinf(enemyPhase[0]); // assume phase stored in radians so no conversion needed

enemyPhase[0] += enemyPhaseVelocity[0] \* tDelta;

}

…then run the demo, you should *see the first enemy* moving up and down.

If you want to update all the enemies, you could repeat the last two lines for each enemy. However, when working with collections we usually use a for loop to iterate through each item. Since the GameObjectCollection ‘enemies’ includes the objectCount property, we know exactly how many enemies to process. Let’s **replace** the last 2 lines we added to myUpdate with the following for loop…

for (int i = 0; i < enemies.objectCount; i++) {

enemies.objectArray[i]->position.y = sinf(enemyPhase[i]); // assume phase stored in radians so no conversion needed

enemyPhase[i] += enemyPhaseVelocity[i] \* tDelta;

}

**Note**: Notice that the inner part of the loop is the same as above, we’re just accessing array element *i*, not 0.

When you run this you should see all 3 enemies move in unison, at the same speed.

**Exercise**: The reason they all move in unison is because their starting phase is the same, and their respective velocities are the same too. Try changing the values in the enemyPhase and enemyPhaseVelocity arrays to see what happens.

This is a good way of updating groups of objects, but has one major problem! What if the number of objects in the group changes? We cannot rely on hard-coded arrays as the size would need to grow or shrink if we added or deleted objects respectively. In a later section we’ll look at object orientation to help solve this problem.

# Getting User Input – Keyboard input

Out of the box the engine allows you to provide user input via the keyboard (we can add mouse and controller input later)! The default keyboard handler in the engine does nothing more than check if the escape key is pressed to quit the game.

However, like the update function, we can provide our own keyboard handling function to handle input in any way we want to. This section looks at how we can do this, how we can use the GLFW library to detect key presses and the best way to respond to these. For this walkthrough we’ll build on the scene setup for “Section 5 Animating Multiple Objects” above. If you’ve not had a chance to complete that section, at the very least have a go at setting up the scene by creating the game objects.

Let’s first create our own keyboard handling function. The return type and parameters must match what is expected by the engine, and in turn the GLFW library (since the engine uses GLFW for windows and input handling). More information about keyboard input with GLFW can be found [here](https://www.glfw.org/docs/latest/input_guide.html#input_keyboard), but the structure of the keyboard handling function is as follows:

**void** key\_callback([GLFWwindow](https://www.glfw.org/docs/latest/group__window.html#ga3c96d80d363e67d13a41b5d1821f3242)\* window, **int** key, **int** scancode, **int** action, **int** mods)

We usually only use the **key** and **action** parameters, but the breakdown of what each parameter does is as follows:

**GLFWwindow\* Window**: The actual window to receive the keyboard event. This is a critical part of event-based systems like Windows – when you type into Word, Word gets the key event. When you type into a Google search box in Chrome, Chrome gets the event.

**int key**: This is the keycode that represents the key that has been pressed or released. GLFW provides constants for all common keys and we’ll use this below.

**int scancode**: A low-level, platform-specific code for a given key that is pressed. For most cases we don’t need to use this for keyboard input.

**int action**: This parameter indicates what’s just happened with the key represented by the ‘key’ parameter. This can be one of 3 constants: GLFW\_PRESS (the key has just been pressed), GLFW\_REPEAT (we’re holding the key down so the Operating System generates these repeat events at regular intervals, and GLFW\_RELEASE (we’ve let go of the key).

**int mods**: The last parameter contains bitflags to indicate what modifier keys were pressed when the key event was created. These can be shift, ctrl, alt etc. More information about the bit patterns used can be found [here](https://www.glfw.org/docs/latest/group__mods.html).

To add our own key handling function, we first create the function prototype (i.e. function declaration) and its main implementation (function definition). At the top of main.cpp in the // Function prototypes section, let’s add the function prototype for our keyboard handler (which we’ll call myKeyboardHandler) – we can put this after the prototype for the myUpdate function we added earlier…

void myKeyboardHandler(GLFWwindow\* window, int key, int scancode, int action, int mods);

Secondly, in the main function, we need to tell the engine we want to use our own keyboard handler, just as we did for the update function. In your scene setup code, find where you set the update function and add the call to setKeyboardHandler after this as follows…

setUpdateFunction(myUpdate);

setKeyboardHandler(myKeyboardHandler);

Finally, at the bottom of main.cpp, just after your myUpdate function, let’s add the bare-bones keyboard handler function implementation (**Hint**: feel free to copy the defaultKeyboardHandler function from Engine.cpp – the starting point of our function will be the same – remember to change the function name though!)…

void myKeyboardHandler(GLFWwindow\* window, int key, int scancode, int action, int mods)

{

// Check if the key was just pressed

if (action == GLFW\_PRESS) {

// now check which key was pressed...

switch (key)

{

case GLFW\_KEY\_ESCAPE:

// If escape is pressed tell GLFW we want to close the

window (and quit)

glfwSetWindowShouldClose(window, true);

break;

}

}

// If not pressed, check the key has just been released

else if (action == GLFW\_RELEASE) {

// handle key release events

}

}

The function is split into two main parts: code that handles the case when the given key is pressed (action == GLFW\_PRESS), and code that handles the case when the given key is released (action == GLFW\_RELEASE). We’ll not worry about repeating key presses (action == GLFW\_REPEAT) since the way we track key presses will avoid the need to do this (more on this below).

Let’s add keys WASD to move our player character around the screen. We’ll also add SPACE to fire, but we’ll come back to this in a later chapter. To start, let’s do a quick test to make sure we can register key presses and releases. For this test we’ll check for the press and release of the ‘w’ key. In your myKeyboardHandler function, go to the first if() branch (that gets called when action == GLFW\_PRESS). Here you’ll find a switch statement to check which key was pressed. At the moment only one case is checked: GLFW\_KEY\_ESCAPE. Let’s add a case for the ‘w’ key after that. The GLFW library contains constants for every key – here we want to check for GLFW\_KEY\_W. Just after the break statement on the GLFW\_KEY\_ESCAPE case, add the following:

case GLFW\_KEY\_W:

printf("w pressed\n");

break;

If you run this, you should see the message “w pressed” appear every time you press the ‘w’ key. Next, let’s check we can detect when the ‘w’ key is released. In the if() branch that deals with action == GLFW\_RELEASE, add the following switch statement…

// handle key release events

switch (key)

{

case GLFW\_KEY\_W:

printf("w released\n");

break;

}

When you run this, you should see ‘w released’ when you let go of the ‘w’ key.

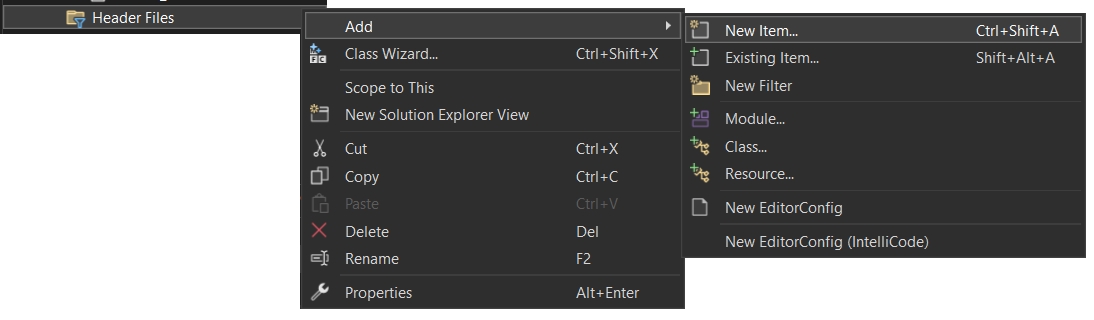
Now we know our key is being detected, let’s actually get it to move the player object setup previously. There’s a temptation to put the actual movement code inside the key event handler, but this has 2 main drawbacks:

* The PRESSED event only occurs once when a key is initially pressed. So, if we put the code to move our player in the GLFW\_PRESSED handler it’d only move once – the PRESSED event will not be (re)generated while the key is held down.
* No problem you say, we have the GLFW\_REPEAT event for when I hold a key down. Yes, you do, but this is only generated at a set frequency, not every frame of our game, resulting in input lag and a jerky movement of the player.

To solve this problem, **all we should do in the key event handler function is just track presses and releases of specific keys**. We put the actual player update / movement code in our update function which does get called every frame. As part of the update function, we check if a key is pressed and perform movement accordingly.

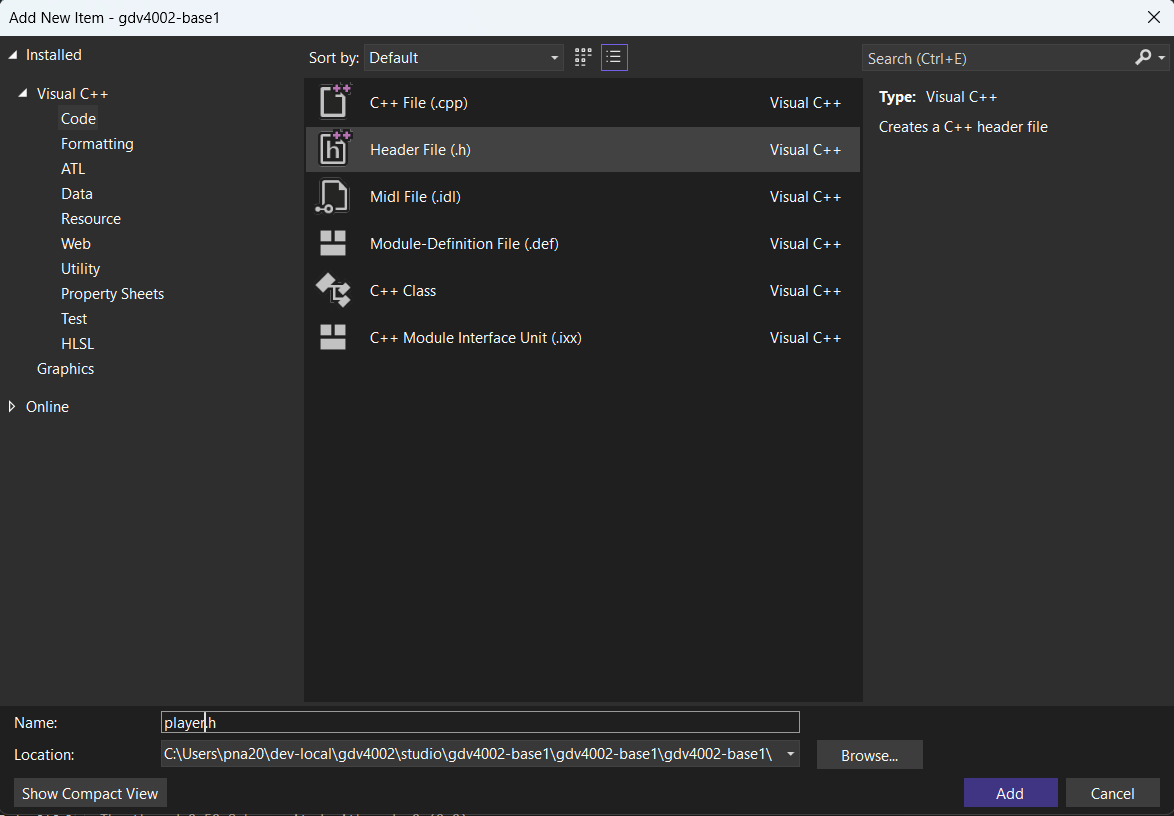
In week 2 we discussed using bits as flags to track if something was true (1) or false (0) as this was more space / memory efficient that using bool variables. Here we’ll make use of C++’s bitset collection to track our keys.

Let’s start off by working out which keys we want to use. For now, let’s setup flags for 5 keys (up (W), down (S), left (A), right (D) and fire (SPACE). We’ll use an enum to define the bit numbers for each key flag. To make this available to other files (which we’ll need in a later chapter), let’s create this enum in its own header file. In the Solution Explorer, right click on the Headers folder and select Add then New Item… in the sub-menu as shown below.



*Figure 6.1. Sub-menus to add a new file*

In the Add New Item… dialog, setup the Keys.h file as shown below…



1. Select code filter

2. Select Header File

3. Name the file Keys.h

4. Make sure the folder is the project folder (should be the default). To check, if you click “Browse…” you should see all the other files including main.cpp – this is the folder we want!

5. Click Add when done!

*Figure 6.2. Setting up a header file*

When the new file is created it just includes the #pragma once directive. After this we’ll add our Key enum declaration…

// Define key constants - these are bit numbers in the following keys bitset

enum Key { W, A, S, D, SPACE };

**Note**: When an enum is defined, each entry is given a numeric (int) value and the sequence starts off at 0 unless otherwise specified. This means our Keys enum entries have the following values:

W: 0

A: 1

S: 2

D: 3

SPACE: 4

Back in main.cpp, include the new Keys.h and bitset headers after we’ve included the Engine.h file…

#include "Engine.h"

#include "Keys.h"

#include <bitset>

Once we’ve included the bitset header, we need to create a bitset variable to store the key press flags (we could use a fixed size integer type as shown in week 2, but the bitset type allows us to have arbitrary collections of bits to suit our application). We’ll set this up as a global variable in main.cpp. Just before the main function add the following…

std::bitset<5> keys{ 0x0 };

There are 5 bits (flags) in our bitset, one for each key we want to track, and all bits are initialised to 0 (false). The bitset can be visualised as follows:

0

bit

1

2

3

4

W

A

S

D

SPACE

*Figure 6.3. Bit flags to track WASD and Space keys*

Let’s go back to our keyboard handlers for the W key and modify them so instead of just reporting to the console, they **set** (for action == GLFW\_PRESS) and **unset** (for action == GLFW\_RELEASE) the relevant bits in the bitset…

So, in the GLFW\_PRESS handler for the W key…

case GLFW\_KEY\_W:

keys[Key::W] = true;

break;

…and in the RELEASE handler for the W key…

case GLFW\_KEY\_W:

keys[Key::W] = false;

break;

At the moment, running the project will track the w key, but we don’t see anything happening yet. To move the player, let’s go to the myUpdate function and add a block of code that checks if the w key flag is set – if it is we’ll move the player by a given velocity. Add the following to your update function…

static float playerSpeed = 1.0f; // distance per second

GameObject2D\* player = getObject("player");

if (keys.test(Key::W) == true) {

player->position.y += playerSpeed \* (float)tDelta;

}

Unlike the enemy move code we added earlier, the player’s position only changes if the w key flag is set (we use the bitset **test** function to check this). Remember, we still need to scale (multiply) the velocity / player speed by the frame time elapsed (tDelta).

Running the project should now allow you to press the w key and while it’s pressed the player will move up the screen!

Let’s add in the ability to press the ‘s’ key to move down. Most of the plumbing is already done, so we only need to add (1) the code to check the ‘s’ key press / release and (2) the code in myUpdate to move the player if the S key is pressed.

In the keyboard handler, add the following code to the GLFW\_PRESS and GLFW\_RELEASE handler blocks respectively…

In the GLFW\_PRESS block, add the following case after the ‘w’ case block…

case GLFW\_KEY\_S:

keys[Key::S] = true;

break;

…and in the GLFW\_RELEASE handler, add the following case after the ‘w’ case block…

case GLFW\_KEY\_S:

keys[Key::S] = false;

break;

In your myUpdate function, just after the if() block where you tested if W was pressed, add the following if() block to test if S is pressed…

if (keys.test(Key::S) == true) {

player->position.y -= playerSpeed \* (float)tDelta;

}

Running this should now mean your player can move up and down!

**Exercise**: Extend your keyboard handler and update functions to allow the player to move left (with the A key) and right (with the D key) as well.

**Problem**: When you move diagonally with up / left pressed together for example, you may notice the player moves marginally faster than just moving in one direction alone? Any guesses why?

**Exercise**: Have a go at changing your update function so instead of moving up, down, left and right, change the movement so:

Pressing A and D will rotate your player counter-clockwise and clockwise respectively. **Hint**: Instead of changing the position you’ll change the player object’s orientation. You’ll also need a rotation speed in angles (stored as radians) per second. In addition, pressing W will move the player character forward in the direction it’s facing. **Hint**: Remember complex numbers and how we can use the exp() function to get a direction vector (x, y) from an angle.

# Using Object-Oriented Techniques for in-game objects

In the above sections, we were able to create game objects using the GameObject2D type, and by adding additional variables (either global or within the update function), we could add other properties such as velocity to the object.

However, there are some real problems with this approach. The data for an object is spread between locations – the player’s orientation and position are stored in the GameObject2D type, but the velocity might be stored as a global variable.

One solution would be to add these variables (also referred to as attributes) to GameObject2D. However, not every game object needs the same attributes. For example, the enemies moving up and down in “Chapter 5: Animating Multiple Objects” used a phase variable to control where they were on the sine wave. This isn’t relevant for the player. Besides, it’s not possible to anticipate ALL the attributes a game developer might think of and put them in GameObject2D!

What we want to do is create our own versions of GameObject2D for each type of object in the game and add the relevant attributes to these. Enter object-orientation and sub-classing!

In this chapter we’re going to do 2 things:

* Create our own game object classes by subclassing GameObject2D and add these to the game scene instead of using the default GameObject2D type.
* When we subclass GameObject2D, we inherit its functions and properties. We’re going to override the update function (which we’ve ignored up until now) to actually do the object update.

To explore these changes, we’ll re-implement the scene from chapter 5, with the player and moving enemies, but using our own classes to represent these in-game elements. We’ll also build on the example setup in chapter 6 using keyboard input so we can move the player around. Let’s start with a “clean slate” and set our main.cpp file to include the key handling code but remove the myUpdate function (yes, we’ll no longer need myUpdate for this example). main.cpp should look something like the following – the keyboard handling code from chapter 6 should stay but the old scene setup code and update function are removed (these aspects are highlighted in yellow below)…

#include "Engine.h"

#include "Keys.h" // setup in chapter 6

// Function prototypes

void myKeyboardHandler(GLFWwindow\* window, int key, int scancode, int action, int mods); // setup in chapter 6

// Globals

// Bit flags to track which keys are currently pressed - setup in chapter 6

std::bitset<5> keys{ 0x0 };

int main(void) {

// Initialise the engine (create window, setup OpenGL backend)

int initResult = engineInit("GDV4002 - Applied Maths for Games", 512, 512, 5.0f);

// If the engine initialisation failed report error and exit

if (initResult != 0) {

printf("Cannot setup game window!!!\n");

return initResult; // exit if setup failed

}

// Setup game objects

// Setup event handlers

setKeyboardHandler(myKeyboardHandler);

// Enter main loop - this handles update and render calls

engineMainLoop();

// When we quit (close window for example), clean up engine resources

engineShutdown();

// return success :)

return 0;

}

void myKeyboardHandler(GLFWwindow\* window, int key, int scancode, int action, int mods)

{

(code from chapter 6 here)

}

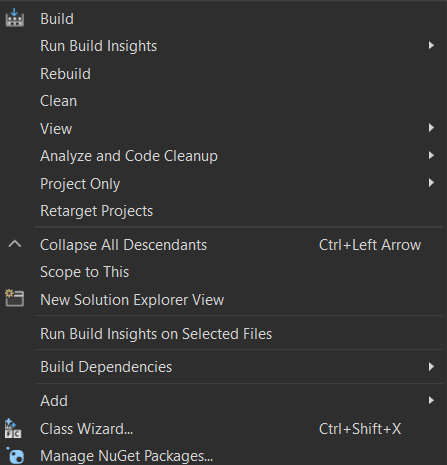
**Note**: Don’t worry about deleting code – remember your repo stores a history of everything you’ve done …

## Adding the Player Class

Let’s add a new class to represent our player. To keep things organised we’ll setup a new folder (also called a filter) in the Visual Studio project. In the Solution Explorer tab (where all your files are shown), right click on the project entry (it’s near the top of the list in bold). In the sub-menu you’ll find the Add option. Select this and a further sub-menu will open. From this select “New Filter…”

A screenshot of a computer

AI-generated content may be incorrect.



A screenshot of a computer program

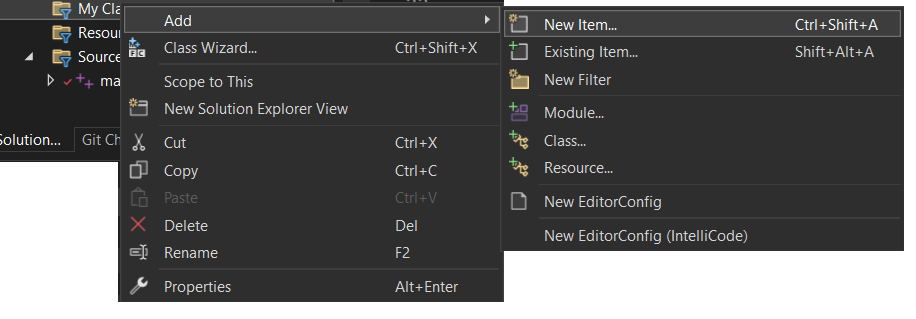
AI-generated content may be incorrect.

*Figure 7.1. Adding a new folder*

A new entry will appear in the Solution Explorer where you can enter the name of the folder / filter. Let’s call this **My Classes**. It’s important to note this does *not* create a folder called “My Classes” in your project folder in Windows Explorer – this is specific to the Visual Studio project. By default, new files are added in the project directory on disk.

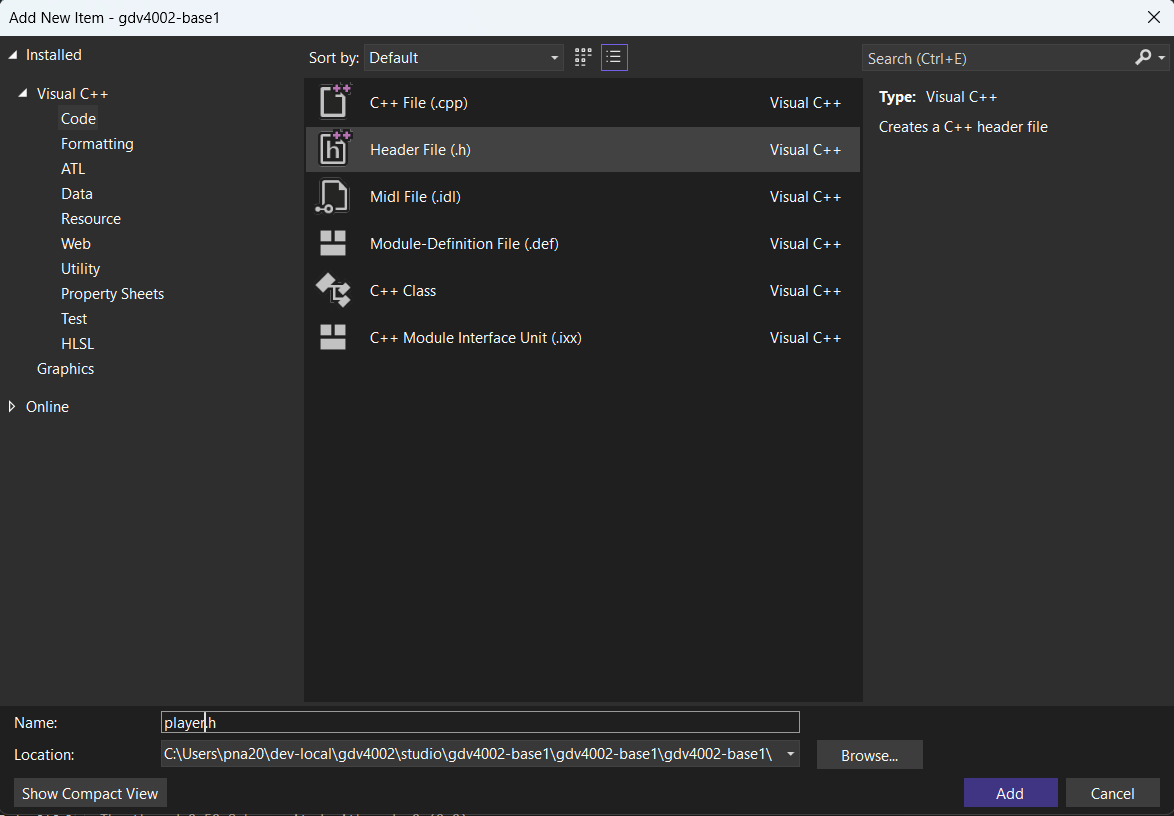
**Note**: For now, we’ll not change this, but as you start to work on bigger and bigger projects, it’s a good idea to create sub-folders to group and organise files on disk too.

Let’s now add the files we need for the Player class. Right click on your new “**My Classes**” folder, select Add in the pop-up menu, and then select “New Item…” as shown below…



*Figure 7.2. Adding a new item*

This will show the **Add New Item** dialog. Set the options to add the Player.h file to the project as shown below…



1. Select code filter

2. Select Header File

3. Name the file Player.h

4. Make sure the folder is the project folder (should be the default) – if you click “Browse…” you should see all the other files including main.cpp – this is the folder we want!

5. Click Add when done!

*Figure 7.3. New file dialog – setting up Player.h*

This will add the Player.h file to the project as shown below.

A black and white text

AI-generated content may be incorrect.

*Figure 7.4. Player.h added to the My Classes folder*

**Exercise**: Using the above approach, add the Player.cpp file to your project. **Hint**: In the Add New Item dialog make sure you’re adding a C++ file, not a header file.

**Note**: When you open the Add item menu, there’s an option to add a Class as well. This allows you to configure aspects of the class such as it’s name and parent class, and creates both the .h and .cpp files. You could use this but here we want to start with empty files and build the classes from scratch. When you’re more familiar with object orientation and C++ syntax for defining classes you can use this option if you wish.

## Setting up the Player header (.h) file

Let’s now create the content of our Player class. The header file includes the class *declaration* – it contains the class name, any parent class, attributes and what functions it contains. The cpp file contains the class *definition*, or *implementation*. Here we put the code for each function declared in the header file. This is a common pattern – you typically want users of your class to know what it’s called and what functions are available (so you make the header files available), but you usually don’t want them to know how the functions are implemented or how they work (you don’t usually distribute cpp files!).

Let’s start by adding in the Player class declaration in the header file. Open the header file add the following…

#pragma once

#include "GameObject2D.h"

class Player : public GameObject2D {

private:

public:

};

A quick overview of what’s going on here…

#pragma once

A header file may be included multiple times in a larger project, but we only need to process it once during the compilation / build process – this tells the compiler exactly that – if you’ve seen the header file before, ignore it please! *This line should already be present in the newly created header file.*

#include "GameObject2D.h"

We’ve included header files before and we need to include GameObject2D.h if we want to create a sub-class if it.

class Player : public GameObject2D {

Here’s the declaration of our Player class. With public GameObject2D we’re saying we want to base the Player class on the GameObject2D class, and the attributes and functions of GameObject2D to be accessible in Player as well.

private:

public:

Anything after the private: statement will be accessible *only within the class* – other parts of the application, such as code in main.cpp will not be able to access these *directly*. Anything after the public: statement will be *visible within the Player class and anything else that uses the class*.

Typically, we put attributes in the private block and functions in the public block. In terms of attributes, since we’ve subclassed GameObject2D, we don’t need to include position, orientation etc. since these are inherited by Player and we can access them as part of the Player class itself. If we look at the keyboard input example in chapter 6, we saw that we needed a playerSpeed variable to control how fast the player can move. In the example shown in chapter 6 this variable was defined in the myUpdate function, but since it’s related to the player, it’s something we can include as an attribute in our Player class…

private:

float playerSpeed;

In terms of functions, classes typically define *constructors* for initialisation, *destructors* to clean things up (free memory, close files etc), *accessor methods* to read / write private attributes, *operators and functions* to process data contained in instances of the class as well as *overriding functions* to replace inherited functionality with the class’s own versions.

For our Player class, we’ll need a constructor to initialise its attributes, but we don’t need a destructor since the class will not be managing any memory, files etc. The constructor needs to provide parameters to initialise all attributes. We can borrow the constructor parameter list from GameObject2D, since we inherit those attributes in the Player class, but we also want to add the playerSpeed parameter that’s specific to the Player class.

We need our Player class to provide its own update code – even though we inherit GameObject2D’s update function, so we want to override this as well. The GameObject2D update function took a single parameter – the time elapsed since the last frame – so we need to do the same in the override.

To declare these functions, add the following code after the public statement…

Player(glm::vec2 initPosition, float initOrientation, glm::vec2 initSize, GLuint initTextureID, float initialPlayerSpeed);

void update(double tDelta) override;

## Creating the Player implementation in the cpp file

With the header file done, let’s look at the function implementations in the cpp file…

The first thing to add to the cpp file are the header includes…

#include "Player.h"

#include "Keys.h"

#include <bitset>

The next part of the code tells C++ we want a bitset for the keys as follows…

extern std::bitset<5> keys;

This looks like the bitset we defined in main.cpp in chapter 6. The extern keyword is important here – it tells C++ we’re not defining an entirely new bitset inside the Player.cpp file, instead we’re saying we want to use a bitset called ‘keys’ but it *can be found elsewhere in the project*. When we later build the project, C++ will locate the bitset in the main.cpp file and use that. So, extern turns this into a sort of proxy to the actual bitset we want to use.

**Note**: Why do this? Well with the update code for the player now moving into the Player class, we’ll still need to know which keys are pressed to do the movement, as we did before. However, we also need to set the key flags from the keyboard event handler in main.cpp. So, two distinct parts of our code need to use the same bitset. In such cases the question of who “owns” the bitset and this where it’s defined crops up. Let’s make a design decision here – we’ll keep the bitset for the keys in main and using the extern keyword we can get the Player to reference that. There are other methods for managing this “who owns what” problem and you’ll explore these further in the Game System Fundamentals module.

The next part of the cpp file will define the constructor – this is done as follows…

Player::Player(glm::vec2 initPosition, float initOrientation, glm::vec2 initSize, GLuint initTextureID, float initialPlayerSpeed) : GameObject2D(initPosition, initOrientation, initSize, initTextureID) {

playerSpeed = initialPlayerSpeed;

}

What we’re doing here is taking all the parameters we need, but we only set the playerSpeed attribute in the body of the function. What about the other attributes we inherited? You’ll notice after the function’s main parameter list there’s a **:GameObject2D(…)** block of code. Since we’ve subclassed GameObject2D, what this bit of code does is call the GameObject2D constructor. We forward the relevant parameters to that function to initialise the inherited attributes. When it’s done, the body of the Player constructor takes over to initialise the attributes we added as part of the Player class – playerSpeed in this case. This is a handy syntax as it allows us to reuse constructors in the parent class without duplicating the setup code in the sub-class constructor.

The final part of our cpp file will be the update function. This will essentially implement the same update code we used for the player in the previous chapter. This will be implemented as follows…

void Player::update(double tDelta) {

// Unlike our myUpdate function, we're already 'in' the player object, so no need to call getObject as we did before :)

if (keys.test(Key::W) == true) {

position.y += playerSpeed \* (float)tDelta;

}

if (keys.test(Key::S) == true) {

position.y -= playerSpeed \* (float)tDelta;

}

if (keys.test(Key::A) == true) {

position.x -= playerSpeed \* (float)tDelta;

}

if (keys.test(Key::D) == true) {

position.x += playerSpeed \* (float)tDelta;

}

}

All we do here is check the key flags and change the player position accordingly. It’s noted in the comment that unlike the myUpdate function where we had to get the player object from the engine, we don’t need to do that here.

## Creating a Player instance in the game scene

That’s all we need to do for our new Player class – let’s go back into main and create a new game object based on this. The first thing we need to do is #include the Player.h file. Next, in the code where we’ve been initialising the game scene, we’ll create a new player object. Before we did everything in one call to addObject – this took the position, orientation etc along with a texture and the engine created a GameObject2D from this. We’re going to use another version (overload) of addObject that takes the name and *pointer to an existing object*.

To use this, we need to create the object and texture ourselves. We’ll create an instance of the Player class with a texture as follows…

GLuint playerTexture = loadTexture("Resources\\Textures\\player1\_ship.png");

Player\* mainPlayer = new Player(glm::vec2(-1.5f, 0.0f), 0.0f, glm::vec2(0.5f, 0.5f), playerTexture, 1.0f);

addObject("player", mainPlayer);

So, what was done in one step is now done in three separate steps, *but we get the benefit of creating our* *own type of game object*!!! If you run the game you’ll see the player, but try moving around. It works! Even though we’ve not got myUpdate anymore, the engine uses its own update function, which calls the update function on all the game objects, including the overriding update function we added to our Player class. The neat thing here is that all the relevant data needed to make the player move is wrapped up (encapsulated) in one place – the Player class – it’s not split up across different cpp files, functions or global variables!

## Creating an Enemy class

In the above sections we created a class to define a Player – it contained the data the player needed and the relevant code to initialise and update the player as part of the game. In this section, we’ll add a new class to define the enemy objects that oscillate up and down.

Firstly, using the method outlined above, create new Enemy.h and Enemy.cpp files in the “**My Classes**” folder within your project. In the header file the Enemy class will inherit from GameObject2D as well, so the outline of the header file will look like the following…

#pragma once

#include "GameObject2D.h"

#include <glm/glm.hpp>

class Enemy : public GameObject2D {

private:

public:

};

When we animated the enemies in chapter 5, we needed two variables for each one:

* + The phase angle to control where we were on the sin wave
  + The velocity of the phase angle – how much the phase angle changes per second, which controls the speed of the enemy movement / oscillation.

In chapter 5 we put these in separate arrays, but here we’re going to store them as float values in the Enemy class. Add the following attributes after the private statement…

float phaseAngle; // in radians

float phaseVelocity; // angle change per second

**Note**: We’re not using an array here because when we create an instance of the enemy class the phaseAngle and phaseVelocity will be for just that single enemy.

For the functions in the Enemy class, we’ll need a constructor and override of the update function again. The Enemy constructor will take the GameObject2D parameters (as did the Player constructor), but also parameters to initialise the phaseAngle and phaseVelocity attributes. To declare the functions, put the following after the public statement in the header…

Enemy(glm::vec2 initPosition, float initOrientation, glm::vec2 initSize, GLuint initTextureID, float initialPhase, float initialPhaseVelocity);

void update(double tDelta) override;

In the Enemy cpp file, we’ll initialise our Enemy objects similar to how we initialised the Player. The following code includes the Enemy.h header and implements the constructor…

#include "Enemy.h"

Enemy::Enemy(

glm::vec2 initPosition,

float initOrientation,

glm::vec2 initSize,

GLuint initTextureID,

float initialPhase,

float initialPhaseVelocity)

: GameObject2D(initPosition, initOrientation, initSize, initTextureID) {

phaseAngle = initialPhase;

phaseVelocity = initialPhaseVelocity;

}

The update function will implement the sin function call we used in chapter 5…

void Enemy::update(double tDelta) {

// Set position based on phaseAngle

position.y = sinf(phaseAngle);

// Update phaseAngle based on velocity \* time elapsed

phaseAngle += phaseVelocity \* tDelta;

}

To setup and use our Enemy class, back in main.cpp include the Enemy.h header file at the top…

#include "Engine.h"

#include "Keys.h"

#include "Player.h"

#include "Enemy.h"

In the scene setup code in the main function, we’ll create three enemies (since we had 3 enemies in the chapter 5 demo). This is done in 3 steps…

1. Load the texture for the enemies. Since all our enemies use the same texture, we only need to load this once.
2. Create the enemy objects
3. Add the enemy objects to the game scene

After the Player object has been created, add the following code to create and add the enemy objects…

// 1. Load enemy texture

GLuint enemyTexture = loadTexture("Resources\\Textures\\alien01.png");

// 2. Create enemy objects

Enemy\* enemy1 = new Enemy(glm::vec2(0.0f, 0.0f), 0.0f, glm::vec2(0.5f, 0.5f), enemyTexture, 0.0f, glm::radians(45.0f));

Enemy\* enemy2 = new Enemy(glm::vec2(1.0f, 0.0f), 0.0f, glm::vec2(0.5f, 0.5f), enemyTexture, 0.0f, glm::radians(90.0f));

Enemy\* enemy3 = new Enemy(glm::vec2(2.0f, 0.0f), 0.0f, glm::vec2(0.5f, 0.5f), enemyTexture, 0.0f, glm::radians(60.0f));

// Add enemy objects to the engine

addObject("enemy1", enemy1);

addObject("enemy2", enemy2);

addObject("enemy3", enemy3);

Now when you run the game you’ll see the enemies moving up and down as before. Again, there’s no myUpdate function and all the data needed for each enemy is wrapped up inside the enemy objects.

## Summary

In this chapter we looked at setting up a game scene using our own objects. This allowed us to tie together the data needed for animation and interaction instead of having it split up across source files and functions or resorting to global variables. Object oriented programming is a standard approach across much of software development, but it is just an option here – we can get the same results with object orientation (this chapter) or without object orientation (see chapter 5).

# Using physics to control a character

In the above sections, we’ve only considered one approach to movement, where we change the position of an object based on it’s (fixed) speed, or velocity and the time elapsed. While this approach has been used for decades and underpins the movement mechanics in older games, it is not physically correct (in other words, our game object does not appear to move in a realistic manner).

While physics concepts have been part of games since their inception, it’s only over the past couple of decades that physics has been considered seriously in game development, from early engines such as MathEngine’s [Karma](https://docs.unrealengine.com/udk/Two/rsrc/Two/KarmaReference/KarmaUserGuide.pdf) physics middleware, to [Havok](https://www.havok.com/havok-physics/) and open-source solutions such as [BulletPhysics](https://github.com/bulletphysics/bullet3). Today game engines such as Unity and Unreal have physics-based systems integrated as standard.

Physics engines are based around a number of key concepts:

* Typically physics systems in games work with **rigid body systems** – that is non-deformable objects that interact with each other. This makes computation (relatively) easier. Soft-body dynamics can be used for cloth, water surface modelling etc. Continuum dynamics models how continuous materials such as fluids, gas etc. work. In games we typically do not model to this level of detail!!!
* When working with physics in games you’ll come across the terms *kinematics* and *dynamics*. Kinematics is focused on the geometric properties of how objects move, their paths of motion etc., and not the forces acting on them. Dynamics does consider the forces acting on them, and since we apply forces to make objects move in games (thrust of a spaceship for example), game physics engines are referred to as **dynamics** systems.
* When you looked at the introduction to Calculus slides, you looked at differentiation and integration. In games we typically apply forces to an object, calculate the acceleration and from there the velocity and position change. This is an application of integration. You would have encountered the integral symbol ∫ which is used for analytical integration. This is where we have “well behaved” functions and interactions that we can model algebraically. However, in games we cannot predict the paths of objects – we don’t know ahead of time what the player will do for example – so we use an approximation of integration called **numerical integration**. The simplest type of numerical integration is called **Euler integration**, which we’ll look at below, but there are many alternatives used in game engines. The heart of any game engine is a numerical **integrator**.
* Physics engines base their models on **Newtonian laws of motion**. These have been established for centuries and for objects moving at “everyday” speeds they hold up well. Unless objects start moving at relativistic speeds (near the speed of light), we don’t need General Relativity!!!

So, using these concepts we’re going to do the following:

* We’re going to apply forces to objects each frame and use Newton’s laws of motion to calculate the acceleration of each object from these forces.
* We’ll use Euler integration to calculate the new velocity and from this the new position of an object.
* **In this example we’ll not consider angular forces (torque) that make objects rotate.**

## Game physics – basic attributes

Let’s go back to the example we developed in chapter 7 (where use used object-oriented techniques to define our own in-game types). Here we’ll make changes to the Player class to add physics properties as summarised above.

To start, each object needs a mass value – how heavy it is. The larger the mass value the larger the forces acting on the object need to be in order to make it move.

Secondly, we need to store the direction the object is moving and by how much. This is the speed, or velocity of the object. In 2D games we need to store how much we move along the *x* and *y* axes, so we can store this as a 2D vector using the same glm::vec2 type we used for the object position.

**Note**: Previously we stored just the speed of the object as a single float value, but this is not sufficient to implement a physics system.

## Adding physics to the Player class

Let’s update the Player class so it has velocity and behaves like a real object might. In the Player.h file, we no longer need the playerSpeed attribute, we’ll replace it with the mass and velocity attributes as shown below…

private:

float mass;

glm::vec2 velocity;

In the constructor declaration in the header file, we specified the player speed as a float as the last parameter. Instead of the speed, we’ll provide the mass of the object (how heavy we want it to be). We can keep the float parameter but change its name from initialPlayerSpeed to mass…

Player(glm::vec2 initPosition, float initOrientation, glm::vec2 initSize, GLuint initTextureID, float mass);

**Note**: We’ll not provide an initial velocity – we’ll initialise this to (0, 0) in the constructor below. You could provide an initial velocity if you wanted your object to be moving when the game starts, or when it’s first created, but we’ll leave this for now.

In Player.cpp we’ll update the constructor to initialise the mass and velocity attributes. Remember to change the last parameter name to mass, but the other parameters stay the same. The body of the function initialises the attributes as shown below…

Player::Player(glm::vec2 initPosition, float initOrientation, glm::vec2 initSize, GLuint initTextureID, float mass) : GameObject2D(initPosition, initOrientation, initSize, initTextureID) {

this->mass = mass;

velocity = glm::vec2(0.0f, 0.0f); // default to 0 velocity

}

The most important changes occur in our update function in Player.cpp. In the update function we implemented in chapter 7 we simply checked for key presses and updated the position using the fixed speed multiplied by the time elapsed.

When implementing a physics system, there are 4 steps we need to implement:

1. Sum all the **forces** being applied to the object. The force is just a vector (glm::vec2) which represents the direction in which the total force is being applied.
2. Using Newton’s , we calculate the **acceleration** vector (also a glm::vec2) as acceleration = total force divided by object mass ().
3. We use an Euler integration step to calculate the new **velocity** based on the acceleration
4. We use another Euler integration step to calculate the new **position** based on the new velocity.

Let’s rewrite the update function! Clear out the function as shown below…

void Player::update(double tDelta) {

}

The first thing we’re going to do is calculate the forces being applied to the player. For our first step, we’ll just add forces based on what key is pressed. For example, if we press ‘w’, we want to apply a force such as (0, thrust), where thrust is some float value that represents the strength of the force being applied. When we press A, S or D we also apply the relevant forces. This is illustrated below.

w

A

S

D

F = (0, thrust)

F = (0, -thrust)

F = (thrust, 0)

F = (-thrust, 0)

*Figure 8.1. Keys and associated force vectors applied to the player*

So, we initialise the total force vector to (0, 0) and based on the key pressed add the relevant force vector to this. The first part of the new update function will look as follows…

glm::vec2 F = glm::vec2(0.0f, 0.0f);

const float thrust = 2.0f;

// 1. accumulate forces

if (keys.test(Key::W) == true) {

F += glm::vec2(0.0f, thrust);

}

if (keys.test(Key::S) == true) {

F += glm::vec2(0.0f, -thrust);

}

if (keys.test(Key::A) == true) {

F += glm::vec2(-thrust, 0.0f);

}

if (keys.test(Key::D) == true) {

F += glm::vec2(thrust, 0.0f);

}

**Note**: We’ve put thrust as a constant here, but feel free to add this as an attribute of the Player class which can be initialised along with the mass in the constructor.

The second step after this is to calculate the acceleration vector. This is calculated as . This is equivalent to as implemented below…

// 2. calculate acceleration. If f=ma, a = f/m

glm::vec2 a = F \* (1.0f / mass);

**Note**: When we store the mass we could also store the inverse, or reciprocal of the mass, that is (). This would avoid a division operator every update (every little saving helps in terms of efficiency). We’ll leave this as an exercise for the workshop!

The third step performs Euler integration where we calculate the new velocity based on the acceleration. This is shown below…

// 3. integate to get new velocity

velocity = velocity + (a \* (float)tDelta);

The fourth and final step calculates the new position based on the velocity. This is what we’ve been doing before when we moved the object, the only difference is that the entire movement is now based on the forces applied to the object.

// 4. integrate to get new position

position = position + (velocity \* (float)tDelta);

Finally, in main.cpp, find where you created the player object in the main function. The last parameter was originally the playerSpeed but remember we changed this to be the mass parameter in the Player class. Let’s keep the numbers simple to start and set the last (mass) parameter to 1.0.

Player\* mainPlayer = new Player(glm::vec2(-1.5f, 0.0f), 0.0f, glm::vec2(0.5f, 0.5f), playerTexture, 1.0f);

Now, if you run the game and press the ‘w’ key you’ll notice it takes a bit of time for the player to speed up in the vertical direction. Let go of the ‘w’ key and player will carry on. If you press ‘s’ to start moving down the player will not move down straight away – it’ll first slow down as you apply the downward force, come to a stop then start accelerating downwards. So we now have physical properties such as velocity and momentum as part of our object’s movement.

**Exercise**: Try experimenting with different mass and thrust values to see the effect.

**Exercise**: As noted above store the thrust as an attribute to be initialised in the constructor. Also store the inverse mass (1/mass) after you set the mass attribute. Hint: You can provide thrust as an additional parameter to the constructor, but you don’t need to pass the inverse mass value – this is calculated from the mass parameter. We call the inverse mass a derived value.

## Adding gravity

There are various other forces we can apply to an object in a game. One common force is drag – this force occurs due to friction or air resistance for example and causes the object to slow down to a stop if no other forces are applied.

Another common force applied is gravity. For planet-based games this is usually a constant force applied downward, towards the ground. Lets’ have a look at adding that here…

Since gravity to not specific to the player, we’ll add this as a global variable in main.cpp. At the top of the main.cpp file, just after the keys bitset variable, add the following…

glm::vec2 gravity = glm::vec2(0.0f, -1.0f);

This creates a vector pointing directly down with size / magnitude of 1.0. Increasing this makes the gravity stronger, decreasing makes the gravity weaker. In Player.cpp, near the top, add the following extern declaration after the bitset extern…

extern glm::vec2 gravity;

When we used extern before we noted that we’re *not defining a new variable* but telling C++ to *locate the variable in another file*.

To apply the gravity force to our player, in the Player update function, just after the key press check (but before step 2 where acceleration is calculated), add the following line to add in gravity to the forces being applied to the player…

F += gravity;

Now run the game – you’ll notice the player automatically accelerates downward unless you press ‘w’ to start accelerating upwards.

**Exercise**: Try experimenting with different gravity values.

## Impulse Forces

Impulse forces occur when there are collisions. An impulse force is something that is applied at one point in time due to a collision, but is not continuously applied (like applying thrust when a key is held down for example).

Let’s add an impulse force to our player. For this example, when the player falls off the bottom of the screen, we’ll add a force to push the player upwards.

When considering the coordinates on the screen, remember we don’t work with pixel coordinates (window coordinates), we work with viewplane coordinates. By default, when we first create the game window in main we can specify the size of the viewplane. The default value is 5.0. The code below shows 5.0 as the final parameter, but you may find this is not shown in your code. Don’t worry if it’s not, it’s still specified as a default parameter value in Engine.h.

int initResult = engineInit("GDV4002 - Applied Maths for Games", 512, 512, 5.0f);

This results in the following viewplane…

x

y

(0,0)

default width of 5.0

-2.5

2.5

-2.5

2.5

*Figure 8.2. Default Viewplane size*

From this we can see the *y* coordinate of the bottom edge is -2.5. This is the value we want to check against when checking if the player “hits” the bottom edge of the screen. Let’s update Player.cpp to check for a collision against this bottom edge and add an impulse force if there is.

First, include Engine.h at the top of Player.cpp (you can put this after the other includes)…

#include "Engine.h"

In the update function, after the line where you added in the gravity force, add the code to check if we hit the bottom edge…

// add impulse force

if (position.y < -getViewplaneHeight() / 2.0f) {

F += glm::vec2(0.0f, 20.0f);

}

Note that when we check the position.y value of the player, we don’t need to hard-code the bottom edge *y* value (-2.5f). We can use the engine function getViewplaneHeight() which returns 5.0 for our example. We then divide by 2 to get 2.5 and then negate it to finally get -2.5. This was why we included Engine.h above, in order to access this function.

The force applied (20.0f) is quite large compared to the other forces we’ve been using. This needs to be sufficiently large to push the player back up into the screen.

Now run the game and checkout what happens when the player hits the bottom edge of the screen!

**Exercise**: Try experimenting with different values for the impulse force.

**Exercise**: Add checks for the other edges of the screen and apply the relevant impulse forces.

# Creating and destroying in-game objects: Particle System example

In this example we’ll explore how to setup a basic particle system. This will introduce the basics of what an Emitter does and how particles are updated, as well as how particles can be deleted / destroyed.

To run this demo, you’ll need the latest version of Engine.h and Engine.cpp (at least version 1.2). You can check the version you have by looking at the top of the Engine.h file. You should see…

#pragma once

// Engine.h ver 1.2

If the version you have is less than 1.2, you can download the latest version from Moodle (in the Additional Resources section on the GDV4002 module page)…

A close-up of a message

AI-generated content may be incorrect.

*Figure 9.1. Engine Source files on Moodle*

This version of the engine should work the same as before but has a couple of bugfixes and additions to make working with particles and object deletion easier!

## Anatomy of a Particle System

Particle Systems are a common feature in video games, often being used to model a large range of natural phenomena such as rain, snow, smoke as well as human-made phenomena such as large arrays of NPCs, bullets etc. – some examples are shown in figure 9.2 below.



*Figure 9.2. Particle Examples.* [*Particle System Overview - Valve Developer Community*](https://developer.valvesoftware.com/wiki/Particle_System_Overview)

There are many features and techniques that underpin modern particle systems, particularly now most particle systems are GPU-based for efficiency. For this example, we’ll not worry about GPU-based particles and only look at two key elements:

**The Emitter**: This is a game object that generates / emits particles. In a game scene an emitter might be seen as an actual object. For example, a jet engine might emit particles to show the thrust, a spaceship might emit, or fire bullet-like particles, a wand might emit lightning or spark-like particles, or a campfire might have wooden logs emitting fire and smoke particles, an example of which is shown below.

A close-up of a campfire

AI-generated content may be incorrect.

Figure 9.3 [Low-Poly Campfire by Ryan Printz on Dribbble](https://dribbble.com/shots/2916432-Low-Poly-Campfire)

However, emitter objects do not necessarily have to be visible objects. In many cases an emitter will be a non-visible object attached to a visible object. This allows us to de-couple the objects we see and interact with from the particle system itself. It also gives us more control. For example, if we have a model of an engine, it’s coordinate will represent where the centre of the engine might be placed. But particles will be emitted form the back-end of the engine. To address this, we’d have a separate (invisible) emitter object attached to one end of the engine object where the particles are emitted from.

An emitter object usually has a position in the game scene as noted above, but it can also have *extent,* or *area*. In the jet engine and campfire examples discussed above, we’d have a *point emitter* since the particles are typically emitted from a single point. However, we might want to emit particles over a wider area – rain or snow for example don’t come from a single point!

**The Particle(s)**: The second key element are the particles themselves. These can vary depending on the application. Since particles are in-game objects they have the usual attributes that game objects have, such as position, orientation, size and a texture / sprite attached. However, particles typically only exist for a short time, and most particles have an *age*, or some criteria that controls when the particle will be destroyed. Age can be useful in many cases as it allows us to alter the appearance of a particle. For example, in the fire particle system shown below, both the fire and smoke particles can change their visual appearance over time.



Fire particles can change form as they get “older”

Smoke particles can change texture, thin out and become more diffuse as they move up away from the fire.

Figure 9.4 [Steam Workshop::Realistic campfire particle](https://steamcommunity.com/sharedfiles/filedetails/?id=1626789417)

For this chapter we’ll build a particle system to create the snowy scene shown below. We’ll create 2 classes – one for the particle then one for the emitter. The emitter’s job will be to create / emit snowflake particles and add them as game objects into the main game scene. As before, we’ll override the update function to process the particles and the emitter.

A screenshot of a computer screen

AI-generated content may be incorrect.

Figure 9.5 – Example Particle System

## Creating a Particle Class

For our particles, we want to create a range of snowflake types that fall from the top of the screen (where they’re emitted) towards the bottom. When they reach the bottom of the screen the particles will be destroyed.

As in the object-oriented examples above, we’ll create a particle class which is a sub-class of GameObject2D. For the behaviour of the particle, we want to use the physics properties we introduced in Chapter 8 for movement only (but not rotation). So, we’ll need a **mass** and **velocity** attribute for this. Though we’ll not use physics for rotation, we’ll still have the snowflakes rotate at a fixed rate. Since the particle will inherit the orientation attribute from GameObject2D, **we need to store how many degrees we want the snowflake to rotate per second**.

In terms of functions, we want to provide a **constructor** to initialise the attributes as well as override the **update** function.

Let’s first setup our particle class. We’ll call this Snowflake to keep it relevant! As before, create a header file called Snowflake.h and a cpp file called Snowflake.cpp – put these in the “My Classes” project folder in Visual Studio if you’ve set that up. The Snowflake header file will declare the class, the attributes and functions we outlined in our description above…

#pragma once

// Model snowflake particle

#include "GameObject2D.h"

class Snowflake : public GameObject2D {

private:

// We'll model physics properties for (linear) movement - that is movement without rotation

float mass;

glm::vec2 velocity;

// ... but we also want the snowflakes to rotate as they fall, so we'll change the orientation attribute but without the physics. We already inherit the orientation attribute from GameObject2D, so we just need to add the 'change in orientation per second' here

float angleChangePerSecond;

public:

Snowflake(glm::vec2 initPosition, float initOrientation, glm::vec2 initSize, GLuint initTextureID, float mass, float angleChangePerSecond);

void update(double tDelta) override;

};

So far, the particle object declaration looks like other objects we’ve created before. Remember, angleChangePerSecond is stored as radians!

Heading into the implementation file (Snowflake.cpp), we’ll first include the Snowflake.h file…

#include "Snowflake.h"

Since our snowflake’s movement will be using the physics techniques we’ve looked at, we’re going to need gravity as a vector to pull the snowflakes towards the bottom of the screen. We’ll use an extern to refer to the gravity vector as we did before…

extern glm::vec2 gravity;

**Note**: Remember, when we use extern we’re not defining a new variable, we’re telling C++ this will be defined somewhere else. In a wider game context, gravity could affect many objects, not just our snowflake particles so it makes sense to define gravity in a more “centralised” location – main.cpp for example.

The first function we need to define is the constructor to initialise the attributes. This follows the pattern we used in previous chapters, and the Snowflake constructor is shown below…

Snowflake::Snowflake(glm::vec2 initPosition, float initOrientation, glm::vec2 initSize, GLuint initTextureID, float mass, float angleChangePerSecond) : GameObject2D(initPosition, initOrientation, initSize, initTextureID) {

this->mass = mass;

velocity = glm::vec2(0.0f, 0.0f);

this->angleChangePerSecond = angleChangePerSecond;

}

A couple of points to note about this:

* Notice the **: GameObject2D(…)** syntax after the parameter list. This is calling the GameObject2D constructor to initialise the inherited attributes which saves us duplicating the code in the Snowflake constructor.
* We noted above we wanted snowflakes to have different textures. This is not the job of the snowflake particle to determine which texture to use – we get given a textureID just like other game objects – it’ll be the emitter’s job to decide which texture / sprite to use when it creates a new snowflake.

The final piece of the Snowflake puzzle is the update function shown below…

void Snowflake::update(double tDelta) {

// 1. Physics bit for movement

// 1.1. Sum forces - only add gravity for now

glm::vec2 F = gravity;

// 1.2. Calculate acceleration

glm::vec2 accel = F \* (1.0f / mass);

// 1.3. Update velocity

velocity = velocity + accel \* (float)tDelta;

// 1.4. Update position

position = position + velocity \* (float)tDelta;

// 2. Non-physics bit for rotation

orientation += angleChangePerSecond \* (float)tDelta;

}

As can be seen, the first part applies the physics system we looked at earlier to change the position of the snowflake, based on the gravity force. The second part just updates the orientation based on the rate angleChangePerSecond – no forces are applied for rotation.

**Note**: There is a temptation to check if a particle needs to be deleted in the update function, but this can potentially lead to problems. The engine uses an iterator to process each game object in the scene. If we delete objects within their update function it’ll invalidate the iterator, and we’d end up with problems. A safer solution would be to decouple the update step from the delete step. Here we’ve discussed update and we’ll look at deleting particles later in the chapter.

## Creating an Emitter Class

Particles, once setup, act as independent objects in the game scene. Much of the work is done in the emitter which tracks when to create particles and then emits them at the appropriate time.

### Emitter Definition

Let’s look at what we need for the emitter. First, since an emitter typically emits particles at a set rate we’ll need to store the following attributes to enable this:

* First, we need a time interval that needs to elapse before the next particle is emitted. For example, if this was set to 1.0 then every 1 second, we’d create a new particle. Typically, this will be small – if we set this to 0.16 then we’d get a new particle every 1/60th of a second. Let’s call this attribute **emitTimeInterval**.
* Second, we need a counter to track the time elapsed since the last time a particle was created so we know when we’ve reached emitTimeInterval and can emit a new particle. Typically, this will be initialised to 0 and every time the emitter’s update function gets called we add tDelta to this. When it reaches emitTimeInterval we create a new particle and reset the counter. Let’s call this counter **emitCounter**.

**Note**: Some systems will emit a particle based on an event like a keypress, but if you want to limit the rate you still need the above attributes.

When we emit a new particle, we also want to give it a unique id so we end up creating game object keys in the sequence “snowflake”, “snowflake1”, “snowflake2”, “snowflake3” etc. We’ll use an attribute particleNumber to count up through the particles – it’ll increase every time we add a new particle to make sure each particle has a unique key / name. We’ll make it an unsigned long long which has a maximum value of 18,446,744,073,709,551,615. Even if we create a particle every 1/60th of a second, it’ll take approximately 10 billion years before we had an overflow error!

We also need to give the particles a texture / sprite. Ideally, we have a range of images we can use so the snowflakes don’t all look the same. To do this we’ll store the texture IDs for each snowflake in an array.

But before we can load textures we need something to load. There’re example snowflake images in the Additional Resources section on Moodle as shown below…

A screenshot of a computer

AI-generated content may be incorrect.

Figure 9.6 – Texture resources on Moodle

If you download and unzip the snow.zip file, you’ll see a folder called “Snow”. Double-click this and you’ll see *another* folder called “Snow”. It’s *this* second “Snow” folder you want to move into your Resources\Textures folder of your project.

Finally, when we create a particle, we want some degree of randomness to them – we want snowflakes to appear at different positions, have a different mass so they fall at different speeds, different textures, rotation speeds and sizes. To do this we could fall back on the old srand and rand functions, but these only produce a random number between 0 and 32767 which we have to remap to our desired range and cast to the appropriate type. This example will look at using random numbers using the newer **<random>** header. We’ll look at how to use this as we go through the example, but in essence we need two objects: a random number engine and a distribution, that is, a range we want the random number to be created in. Typically, we have a number of different distributions for different uses. For the particle emitter, we’ll want to store the following random number-related objects:

* The random number engine
* A distribution for the sprite image – since there are 8 images in the examples noted above, this distribution would range form 0 to 7 (inclusive).
* A normalised distribution between -1 and +1. We can use this for the position of the new particle – we just scale the random value by the size of the view plane.
* A distribution for the snowflake / particle mass. This will typically be very small, and you can experiment with different values, but from testing done this can be between 0.005 and 0.08.
* A distribution for the scale, or size of each snowflake. Again, this can be found by experimenting, but a range between 0.1 and 0.5 can work.

Since the Emitter will inherit from GameObject2D, we also inherit position, orientation, size and textureID. However, since the emitter will not be rendered, we only need the position and scale attributes to determine the area new particles will be created in. We’ll revisit this below.

In terms of functions, we need a **constructor** to set the position, size and emit time interval attributes (other attributes can be passed as parameters but for this example we’ll hard-code the values for now). We also need to override the update function since this is where we count the time elapsed and create a new particle if needed. We’ll also override the render function. The demo engine has no mechanism to switch rendering on and off for specific game objects, but by overriding the render function and essentially doing nothing within the function, nothing will be drawn!

To bring this together, our Emitter header file will look as follows…

#pragma once

#include "GameObject2D.h"

#include <random>

class Emitter : public GameObject2D

{

private:

float emitTimeInterval;

float emitCounter;

unsigned long long particleNumber; // monotonically increasing particle index / number - used to set key

GLuint snowflakes[8];

// Random number generator

std::mt19937 gen;

// Random number distributions

std::uniform\_int\_distribution<int> spriteDist; // random integer for particle sprite selection

std::uniform\_real\_distribution<float> normDist; // -1 to 1

std::uniform\_real\_distribution<float> massDist, scaleDist;

public:

Emitter(glm::vec2 initPosition, glm::vec2 initSize, float emitTimeInterval);

void update(double tDelta) override;

void render() override;

};

### Emitter Implementation

In terms of the Emitter implementation in the Emitter.cpp file, we first include the relevant headers and define the constructor function body…

#include "Emitter.h"

#include "Engine.h"

#include "Snowflake.h"

using namespace std;

Emitter::Emitter(glm::vec2 initPosition, glm::vec2 initSize, float emitTimeInterval) : GameObject2D(initPosition, 0.0f, initSize, 0) {

}

The first thing we want to do inside the constructor is set the timer data…

this->emitTimeInterval = emitTimeInterval;

emitCounter = emitTimeInterval;

We set the emitCounter to emitTimeInterval so first time through the emitter’s update function we emit a particle. If we set emitCounter to 0 then we’d need to wait emitTimeInterval seconds before the first particle was created.

The next step in the constructor is to initialise the particleNumber value – this starts at 0 and increases by 1 each time a particle is emitted. We’ll append this to the particle object name (key) when it’s created.

particleNumber = 0;

The next step is to load the textures for the particles. There are 8 snowflake textures in the example folder on Moodle, so we could just call loadTexture 8 times. However, since each texture image file is numbered and follows the structure “snowflake<number>.png”, why not load them inside a loop as shown below?

for (int i = 0; i < 8; i++) {

string path = "Resources\\Textures\\Snow\\snowflake" + to\_string(i+1) + string(".png");

snowflakes[i] = loadTexture(path.c\_str());

if (snowflakes[i] > 0)

cout << "successfully loaded texture " << path << endl;

else

cout << "failed to load texture " << path << endl;

}

The next step is to setup our new random number generator. First, we create a random\_device object and use this to initialise the random number engine…

// Obtain a seed for the random number engine

random\_device rd;

// Standard mersenne\_twister\_engine seeded with rd() - mt19937 is a high-quality pseudo-random number generator

gen = mt19937(rd());

There are a range of engines, but the mt19937 engine performs well under most circumstances! The above code performs a similar step to srand when using the older approach.

Next, we setup the different distribution objects for each random range we want to use…

spriteDist = uniform\_int\_distribution<int>(0, 7);

normDist = uniform\_real\_distribution<float>(-1.0f, 1.0f);

massDist = uniform\_real\_distribution<float>(0.005f, 0.08f);

scaleDist = uniform\_real\_distribution<float>(0.1f, 0.5f);

The neat thing about this approach is that we can get random numbers in the correct range and in the correct number type without casting and remapping which we’d need to do when using the older rand function. Once this has been done the constructor for the emitter is complete – we have timing data, textures and random numbers setup.

The next step is to implement the overriding update and render functions. Let’s start with render – that’s an easy one…

// override render but do nothing - we'll not render anything for the emitter

void Emitter::render() {

}

We do nothing here since we don’t want any on-screen representation of the emitter – it’ll emit snowflakes only!

**Note**: In the introduction above we discussed the fact that emitter objects are usually distinct but attached to the visual in-game object that appears to be generating the particles – a wand emitting lightning, campfire logs emitting flames and smoke. However, this is not mandatory – it’s a design decision. The properties of an emitter discussed here could just as easily be part of a visible in-game object too.

For the update function, we first use the time elapsed parameter (tDelta) to update the emitter counter. If the counter reaches or exceeds emitTimeInterval we go ahead and create a new particle. The outline of this is shown below…

void Emitter::update(double tDelta) {

emitCounter += (float)tDelta;

while (emitCounter >= emitTimeInterval) {

// decrease emitCounter by emitTimeInterval - don't set to 0 as this would ignore the case where multiple particles are needed.

emitCounter -= emitTimeInterval;

// Create new particle

}

}

Before we get into creating a particle, why have a while loop? For most of the time, tDelta will be small enough that emitCounter passes emitTimeInterval and we emit a single particle – this is illustrated in the timeline shown in figure 9.7.

.

*emitTimeInterval*

*Create particle at these time points*

*Game timeline*

*tDelta*

*Typically, tDelta is small enough compared to emitTimeInterval*

*That for any given frame we usually only create one particle.*

Figure 9.7. Typical timeline where 1 particle emit event occurs in a given frame.

However, if our system is running slow, or for a given iteration of the game loop the system slows down, tDelta might be much larger resulting in emitCounter exceeding multiples of emitTimeInterval – in other words, we’ve skipped the creation of multiple particles due to the slowdown. This is shown in figure 9.8 below.

*emitTimeInterval*

*Create particle at these time points*

*Game timeline*

*tDelta*

*However, me may have a situation where the system is slow or slows down for a given frame and tDelta is much larger. To keep up with the number of particles needed, we may need to emit multiple particles in a given update function.*

Figure 9.8. Larger tDelta means multiple particles may need to be created in a given call to update.

The while loop allows us to count back through how many times we passed emitTimeInterval in the current frame – that’s how many particles we need to add.

**Note**: This is all about consistency across systems – we could just create one particle but then user-experience would be different. If we wanted a heavy snowstorm but slower systems reduced the particle creation rate we’d get a different visual effect.

When we come to create a particle, we want to create random numbers for the position, size, mass, rotation speed and sprite / texture to use. This is done as follows. After the // Create new particle comment within the while loop add the following…

float x = position.x + normDist(gen) \* size.x;

float y = position.y + normDist(gen) \* size.y;

float scale = scaleDist(gen);

float mass = massDist(gen);

float rotationSpeed = glm::radians(normDist(gen) \* 45.0f);

int spriteIndex = spriteDist(gen);

This is where the new random number system becomes useful. We can directly generate random numbers in the desired range and of the desired number type without normalisation, remapping or casting as we did with rand.

**Note**: When the position (*x, y*) is created, we use the “size” of the emitter game object to define the area in which particles will be created – this area spans the width of the on-screen game window.

Once we have the properties setup, we can create a snowflake object as follows…

Snowflake\* s1 = new Snowflake(glm::vec2(x, y), 0.0f, glm::vec2(scale, scale), snowflakes[spriteIndex], mass, rotationSpeed);

Next, we’re ready to build the key, or name of the object. As noted above, we’ll start all particles with the name “snowflake”. This allows us to treat all snowflakes in the engine as a collection of objects. We need to give the particle a unique id number from the emitter – this is where the particleNumber attributes comes in. We append this to the “snowflake” string and increment it for the next particle we create. This is shown below…

string key = string("snowflake");

if (particleNumber > 0) { // first name in collection must not be numbered if using this approach

// add value so unique anyway - not using engine mechanism

key += to\_string(particleNumber);

}

particleNumber++;

Once we have the key, we simply add the particle object to the engine…

addObject(key.c\_str(), s1);

**Note**: The engine has its own internal mechanism for adding an index to the end of objects with the same name, but this does not properly support object deletion. Where we have a system that generates multiple objects that will be deleted at some point, we need our own name mechanism to avoid problems. This is why the emitter uses the particleNumber attribute as part of the object’s name.

**Note**: You may want particles to be created when some event occurs – like pressing a key. In this case you might not want to use the emitCounter mechanism and only emit a particle when a key has been pressed. Alternatively, you could combine these and use the emitCounter to implement a kind of emit/fire rate mechanic.

## Using an Emitter

For the snowflake example, setting up the emitter is straightforward. In main.cpp, include the Emitter header file first…

#include "Emitter.h"

Before we head into the main function, remember we wrote our particle update function to use physics, and as part of this we referenced a gravity variable. We need to define that here. Just after your function prototypes in main.cpp, add a new global variable as follows…

// Global vars

glm::vec2 gravity = glm::vec2(0.0f, -0.005f);

In the main function, where the scene objects are setup, create and add a new emitter object as follows…

Emitter\* emitter = new Emitter(

glm::vec2(0.0f, getViewplaneHeight() / 2.0f \* 1.2f),

glm::vec2(getViewplaneWidth()/2.0f, 0.0f),

0.05f);

addObject("emitter", emitter);

When we calculate the emitter position, we set its *x* position to 0 and its *y* position *just above* the top of the view plane. This ensures new particles are created just outside the visible region of the view plane and avoids a popping effect which would occur if they suddenly appeared on-screen when being created.

If you run this, you should see snow particles falling from the top of the screen! We didn’t need to add these in main, the emitter takes care of this for us!

**Note**: You may do this differently if your emitter has been merged in with another game object (such as a spaceship, wand, campfire etc.) as you’d not need to create a separate emitter object as done here.

## Deleting Particles

The final piece of the puzzle is to delete particles. To do this correctly a number of things are in place:

* We have version 1.2 of the Engine.h and Engine.cpp source files
* When we create a particle, we use a particleNumber counter to give the particle a unique ID (separate from the mechanism in the engine).

Why use Engine version 1.2 (or above)? Aside from bug fixes, we want to be able to use the engine’s built-in update mechanism to update all the game objects (including the emitter and particles), but once that’s done call a function to check through and delete any particles that need deleting.

With engine version 1.2, we can provide our own update function but tell the engine to call this *after* it’s processed all the game objects. Our “update” function will be responsible for checking and deleting objects. In other words, we get to call *both* the built-in update mechanism as well as our own update functionality. In older versions of the engine, you could use the built-in mechanism or your own update function, but not both!

For the snowflake example, we’ll provide our own “update” function to manage the deletion of particles. Let’s call this function deleteSnowflakes. First, we add the function prototype…

void deleteSnowlakes(GLFWwindow\* window, double tDelta)

In the main function, before we call engineMainLoop, let’s tell the engine about the new function…

setUpdateFunction(deleteSnowlakes, false);

The second parameter is new for engine version 1.2. When false we’re telling the engine to call deleteSnowFlakes **after** it’s processed the update on all the game objects. When true, we’re saying we want to completely replace the built-in update and just use our own function (this was the behaviour prior to version 1.2).

The implementation of the deleteSnowflakes function gets the object collection based on the shared key “snowflake” (remember, all snowflake objects start with this in their name so the engine sees them as a single collection). The function then iterates through the particles, and if the condition to delete is met the object / particle is deleted. The condition to delete is when the *y* coordinate of the snowflake passes the bottom edge of the screen. The implementation is shown below:

void deleteSnowlakes(GLFWwindow\* window, double tDelta) {

GameObjectCollection snowflakes = getObjectCollection("snowflake");

for (int i = 0; i < snowflakes.objectCount; i++) {

if (snowflakes.objectArray[i]->position.y < -(getViewplaneHeight()/2.0f)) {

deleteObject(snowflakes.objectArray[i]);

}

}

}

If you run the code the snowflakes will behave as before, but they will be deleted when they reach the bottom of the screen.

So why is it safe to delete the particles here and not in their own update function? This isn’t an engine issue, it’s an issue with iterators in the C++ Standard Template Library (STL). Iterators are a convenient way of looping through collections, but care must be taken when changing the collection (deleting objects in this case) otherwise iterators can become invalid and lead to crashes. This is what would happen if you tried to delete an object in the particle’s own update function. The iterator used to access and call update on the particle would become invalid.

There are other ways to approach this in game engine design, but this is something you’ll revisit later in your course!

**Note**: If you’re not using object-oriented approaches you don’t need to worry about using the update function on objects – all your update code will be in your own update function. The key point is that the delete functionality shown above should happen at the end of your update function.

# Controlling the view

In the previous sections we’ve been working with coordinates around the origin, despite the fact we’re using a window of size 1024 pixels across by 1024 pixels vertically. This is because in computer graphics, we work with geometry that’s “drawn” into a **viewplane** before it’s mapped to the on-screen window (the **viewport**) and turned into pixels (this process of turning geometry into pixels is called rasterisation – every GPU in existence has this built in).

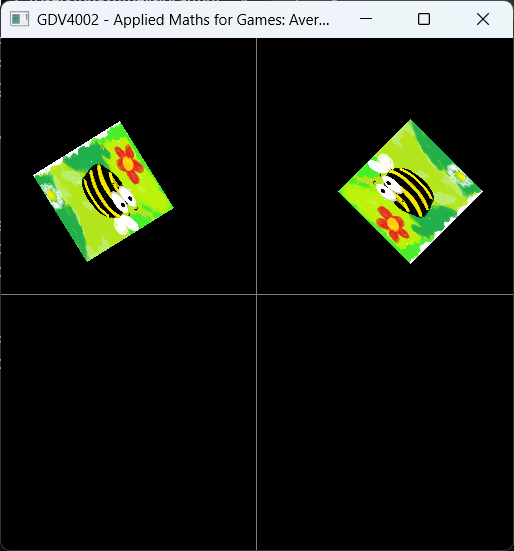
x

y

(0,0)

default width of 5.0

Viewplane



x

(0,0)

y

Viewport (in window)

-2.5

2.5

-2.5

2.5

*Figure 10.1. Viewplane to window (viewport) mapping*

The view plane has a default with of 5.0 (the height is calculated automatically based on the aspect ratio of the window – that is the ratio of the width to the height). This means the visible region of the viewplane surface is bounded by the *x* coordinate range of -2.5 to +2.5 and similarly a *y* coordinate range of -2.5 to +2.5.

**Note**: Above it states the *visible region* of the viewplane. Technically the viewplane extends to infinity, and anything drawn outside, or partially overlapping the visible region is *clipped* so the parts of objects outside the visible region are not drawn.

You can change the size of the visible region either when the main window is created or at any time by using the setViewplaneWidth function. Let’s look at both approaches.

## Setting the viewplane size at window creation time

To set the viewplane size at creation time, at the very start of the main function (in main.cpp) we had the line…

int initResult = engineInit("GDV4002 - Applied Maths for Games", 512, 512);

This uses a default parameter for the viewplane width – this is actually equivalent to saying…

int initResult = engineInit("GDV4002 - Applied Maths for Games", 512, 512, **5.0f**);

We can change the last parameter to a suitable range that works for you. Let’s see what a scene looks like with the default width of 5.0 and with the changed width of 10.0…

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

*Width = 5.0*

*Width = 10.0*

*Figure 10.2. Different viewplane sizes*

As can be seen, when the view plane size gets bigger it appears we zoom out of the scene.

**Exercise**: Try setting the view plane size to different values to see how we can also zoom into a scene.

## Setting the viewplane after the window has been created

If we wanted to change the view plane size during a game, not just at initialisation time, we could do this with the setViewplaneWidth function. For example:

setViewplaneWidth(10.0f);

This allows us to update the viewplane mid-game – useful if you want to control how much of the scene is visible via a camera zoom effect for example.