V 1.1

THE PLAYBUFFER

THE PLAYBUFFER

SINGLE HEADER C++ LIBRARY

// MANUAL AND TUTORIAL //

SUMO DIGITAL

THE PLAYBUFFER

PlayBuffer is a 'Single Header Library' which allows you to program 2D games for **Windows** using **C++** and **Visual Studio**. Single Header Libraries include all their content in one file so that they are easier to incorporate into your own programs. Ours is called "Play.h" and it provides basic functionality which is designed to help you to create 2D games of the kind that were common in the 1990's (platform games, arcade games, etc).

C++ is the 'native' programming language used in game development and a highly sought-after programming skill in many different technology industries. Yet it is relatively hard to learn, and not the right place to start if you've never done **any** programming before. We'd recommend that you get reasonably confident with another language (such as C#, Java or Python) before trying to learn C++ or use the **PlayBuffer**.

The **PlayBuffer** library has been designed to work with Microsoft's **Visual Studio** development environment. This is often already installed in IT labs at schools, colleges and universities and the Community edition version can be obtain for free for home use. This tutorial[1] is quite useful in terms of getting everything set up and working. It was written for **Visual Studio 2017**, but it works fine for 2019 too.

This document describes the **PlayManager** interface, which is the best starting point for using the **PlayBuffer** library if you're relatively new to C++. This includes a **GameObject** manager which is aimed at making C++ game development easier for A-Level or college students. First-year undergraduates or college students can use the same system to explore an object-oriented approach to game development by creating their own GameObject class hierarchies.

[1] https://tutorials.visualstudio.com/cpp-console/intro

THE PLAYMANAGER INTERFACE

PlayManager provides a simple functional interface to the **PlayBuffer** library for learning C++. The C++ language can be challenging to get to grips with, so the **PlayManager** was designed to be used without worrying about some of the more advanced features like memory management or object-orientation. It provides a way of introducing the basic vocabulary and grammar of C++ in a more engaging way than typical beginner tutorials based solely on outputting text to the console. The interactive visual feedback you get from developing games also supports the process of learning how to program, and the lightweight library makes projects quick and responsive to edit and debug. It doesn't rely on resource-hungry game engines or external libraries to make it work and runs reliably on lower specification machines. So, what are you waiting for?

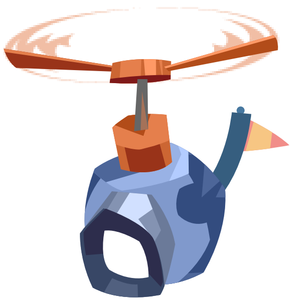
This document is split into two main parts:

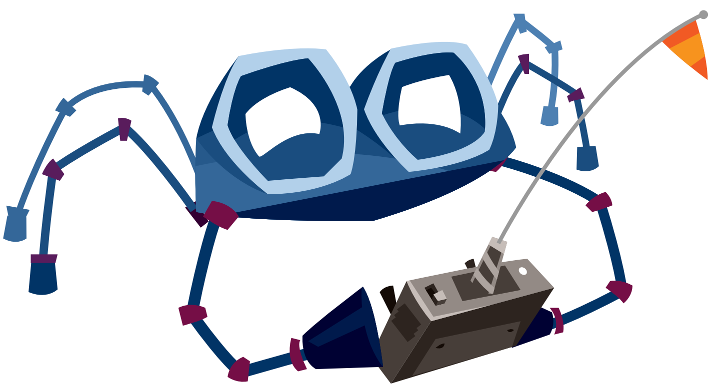
PART 1: GETTING STARTED

This is a complete tutorial for making a very simple shoot-'em-up game using the **PlayManager** interface. It builds a complete game starting from a "Hello World" example and introduces the main areas of functionality provided by the **PlayBuffer** library.

PART 2: COMMAND REFERENCE

The second part of the manual is a complete reference of all the **PlayManager**'s functions for when you need to use them in your own games.





CONTENTS

SPYDER

SPYDER

GETTING STARTED

C++

The **PlayManager** was specifically created to provide an engaging way of learning C++, but this manual can't provide a complete introduction to the language on its own. If you need more support then there are plenty of C++ courses that you can access online, including some free ones (e.g., https://www.sololearn.com/Course/CPlusPlus/).

However, for those who prefer to learn by experimenting rather than following a set curriculum, this section provides a quick overview of the language syntax and structure that you'll need to apply to make your first **PlayManager** game. It assumes you already have some programming experience in another language but should be enough to get you up and running if you're happy to have a go!

So, let's start our whirlwind tour of C++...

COMMENTS

Comments are parts of the code which are ignored and only exist to provide additional information to human beings! Anything which follows a double backslash '//' until the end of the same line, is interpreted as a comment in C++.

```
// This is a comment and is ignored
```

CURLY BRACKETS OR BRACES

Curly brackets are used to mark the beginning and end of groups of statements in C++. These must match so that every opening curly bracket '{' is matched by a closing one '}'. Un-matched curly brackets can produce confusing error messages, so be careful about where you place them!

```
{
    // Some code goes inside here
}
```

BASIC TYPES

C++ has a range of basic data types for its variables, for example:

- Integer types (int) store whole numbers (e.g., 5, 1, 0, -8).
- Floating point types (float) store decimal values (e.g., 1.7, 5.6669, -0.1345). Floating point values in code should end with the letter f (e.g., 1.7f, 5.6669f, -0.1345f).
- Boolean types (bool) are either true or false.
- Character types (char) store text characters (e.g. 'A', 'p', '£') and char* types 'point' to longer sequences of text characters (e.g. "Hello World").

SEMI-COLONS

Expression statements in C++ end in a semi-colon. It's a bit like a full-stop and indicates that the statement has finished. Missing off a semicolon is a common error for new C++ programmers!

```
{
    int var = 0; // create a new integer called 'var' and assign it the value 0
}
```

Note that we're initialising the variable's value to 0 when we create it. It's important to initialise your variables to have a specific value as they otherwise contain random data which could give your variable a different value each time you run your game!

VARIABLE SCOPE

Variables which are declared inside of a function are said to have 'local' scope, which means they are only accessible from within that same function (and after the point at which they were declared). Variables which are declared outside of any function are said to have 'global' scope and can be accessed from within any function (after the point they were declared). However, global variables are not encouraged in C++ and they should be used very sparingly, and once you become more proficient in C++ it's usually best not to use them at all.

```
int globalVariable = 4; // variable has global scope

void DoSomething()
{
   int localVariable = globalVariable; // variable has local scope
}
```

ARRAYS

Any data type can also be declared and accessed as an array. An array is a series of elements of the same type accessed using an index, which is placed inside square brackets ([ ]). In C++ the indices start from 0 and range up to one less than the size of the array. For example:

```
{
    int bonus[4] = { 1000, 2000, 4000, 8000 };
    score = score + bonus[0]; // adds 1000 to the score
    score = score + bonus[3]; // adds 8000 to the score
    score = score + bonus[4]; // error – index out of range!
}
```

FUNCTIONS

Functions definitions begin with their 'return type' (output) and are followed by the function name[2]. If they don't return anything then their return type is set to 'void'. The subsequent parentheses (curved brackets) mark the beginning and end of the 'parameters' (inputs) for that function and each parameter has its own type and name.

The next line contains an opening curly bracket ({) which marks the start of the code for this function (the function 'body'). This function concludes with a return statement which

---

[2] In the **PlayBuffer** we stick to the convention that functions start with a capital letter and variables start with lowercase.

returns the output in the appropriate type. The end of the function is formally marked by the closing curly bracket (}), but any code after the return statement will not get executed!

```
int Bigger( int a, int b ) // integer return type and two integer parameters
{
   if( a > b )
   {
      return a; // integer is returned
   }
   else
   {
      return b; // integer is returned
   }
}
```

The code to call the same function would look like this:

```
{
   int biggest = Bigger( 7, 4 ); // 7 is returned and assigned to the variable 'biggest'
}
```

C++ needs to know about a function before it can be called. This could be achieved much of the time by simply putting your function definitions at the start of files, but it's not always convenient to do this. Fortunately, we can also declare a function without defining it, that way C++ knows how to make use of the function before it knows how it works inside!

```
int Bigger( int a, int b ); // integer return type and two integer parameters
```

You'll need to be careful that any function declarations match the equivalent function definition in terms of their return type and arguments, or you will get compile errors.

REFERENCES

Imagine that we wanted to write the following code:

```
{
    int x = 4;
    int y = 10;
    SwapValues( x, y );
    // x and y should now have swapped values, but have they?
}
```

We might assume this would work, but C++ makes copies of variables which are passed into and out of functions, so changing a variable inside a function doesn't change the original variables used as arguments, but just local copies of them:

```
void SwapValues( int a, int b ) // Two integer parameters are passed 'by value' (copied)
{
   int temp = b;
   b = a; // b is a local variable which holds a copy of the value of y
   a = temp; // a is a local variable which holds a copy of the value of x
   // The values of x and y have NOT changed in the calling function!
}
```

However, making a very small change to the function can fix this. Adding the 'ampersand' symbol in front of the parameter name, indicates that this should be treated as a 'reference' to the original variable rather than a copy. Now the function changes the original values and swaps their values around in the calling function, as originally intended.

```
void SwapValues( int &a, int &b ) // Two integer parameters are passed 'by reference'
{
    int temp = b;
    b = a; // b is a reference to the original y variable: changing b changes y
    a = temp; // a is now a reference to the original x variable: changing a changes x
    // The values of x and y HAVE changed in the calling function!
}
```

References can also be used as return values from functions when it is necessary to return a reference to something rather than just a copy of it.

STRUCTS

Sometimes simple types are not enough, so C++ allows you to group types together to form data structures (structs). The parts of the structure are called data 'members' and should include default values which are used to initialise any variables created of this new data type.

```
struct Point2f
{
    float x = 0.0f; // the default value for x will be 0
    float y = 0.0f; // the default value for y will be 0
};
```

Once defined these can be declared like other variables, but with some additional ways to initialise and access their members.

```
{
    Point2f pos[3] = { { 5.0f, 3.3f }, { 4.2f, 1.0f }, { 1.0f, 0.2f } };
}
```

ENUMERATIONS

Integers (int) can be used to represent all sorts of meanings in code. For example, we might use the value 0 to represent facing left and 1 for facing right. We can define a 'Facing' enum which assigns meaningful names and automatically assigns a value.

```
enum Facing
{
    LEFT = 0,
    RIGHT
};
```

Hello World

If you don't already have **Visual Studio** installed on your PC then do it now. You need **Visual Studio 2019** and the **Community Edition** for home use. This tutorial is quite useful in terms of getting everything set up and working: https://tutorials.visualstudio.com/cpp-console/intro: it was written for **Visual Studio 2017**, but it works fine for 2019 too.

The quickest way to get started with the **PlayManager** is to begin with a copy of the "HelloWorld.sln" project provided. If you've downloaded this a .zip file then don't forget to extract the files first. Open the solution (.sln) file in Visual Studio (double-clicking on the file should normally do that). Once it has loaded press the Green "Play" button on the menu bar (marked "Local Windows Debugger" in figure 1 below). After short pause (as Visual Studio compiles the code) the orange window shown in figure 2 should appear.

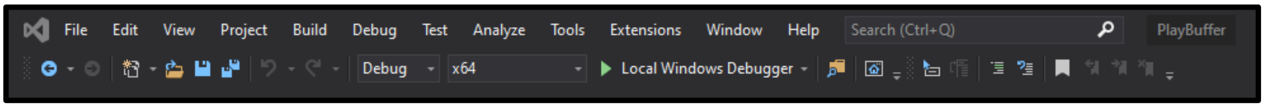


Figure 1: The Visual Studio Menu Bar (Editing).

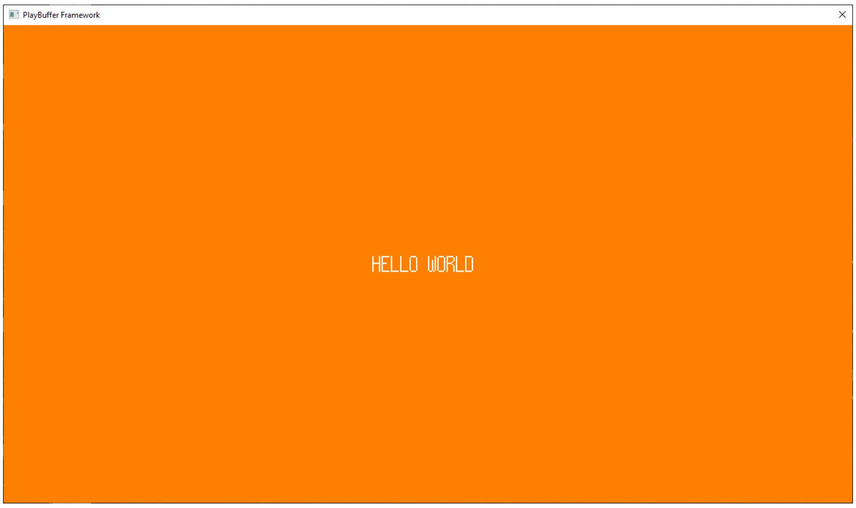


Figure 2: The executable from the "Hello World" solution.

When the program is running the menu bar will change to show the red "Stop" button which you can press to end the program. Normally you can also end the program by pressing the X button in the top, right corner of the window, or by pressing the escape key, but if those fail then the red stop button should always work.

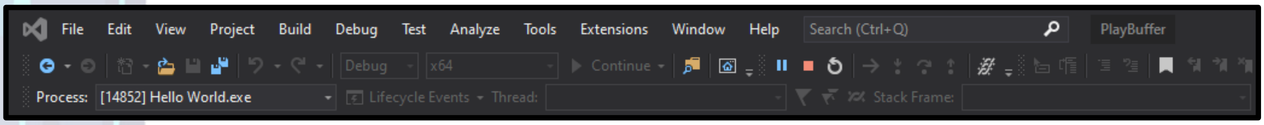


Figure 3: The Visual Studio Menu Bar (Running).

The Main Event

Double click on MainGame.cpp in the **Solution Explorer** to open the code editor. Near the top of the file you will find the most important three lines of code in all **PlayManager** programs! These tell the compiler to implement the **PlayBuffer** library and activate the **PlayManager:**

```
#define PLAY_IMPLEMENTATION
#define PLAY_USING_GAMEOBJECT_MANAGER
#include "Play.h"
```

Listing 1: *The typical way to include the **PlayBuffer** library in your projects.*

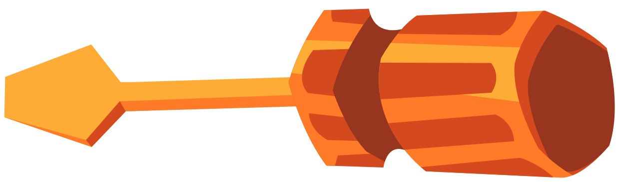
There are three functions which are central to the way the **PlayBuffer** works and you'll need to have all three implemented in your program for it to operate. If you've done any C++ coding before you may recall that programs typically have a 'main()' function which defines the 'entry point' to the program (where it starts). PlayBuffer programs have a MainGameEntry() function instead, which follows the pattern below:

```
int DISPLAY_WIDTH = 640;
int DISPLAY_HEIGHT = 360;
int DISPLAY_SCALE = 2;

// The entry point for a Windows program
void MainGameEntry( PLAY_IGNORE_COMMAND_LINE )
{
    Play::CreateManager( DISPLAY_WIDTH, DISPLAY_HEIGHT, DISPLAY_SCALE );
}
```

Listing 2: *A typical **MainGameEntry()** function in a **PlayManager** program.*

The variables before MainGameEntry() are global variables which store the dimensions of the display buffer. Because they are declared outside of any function, they have "global scope" and can be accessed throughout the program and time we want to know the size of the display.

The MainGameEntry() function uses a 'macro' (in purple) for its parameters. Macros like this just replace one thing with something else, which allows us to hide a bit of complexity here. You don't need to use command line arguments for most PlayBuffer programs, and this simplifies things until you do.



Inside MainGameEntry() calls Play::CreateManager() to set up the display buffer and window at the desired size and resolution (Listing 2: Line 12). We will need to use this Play:: syntax to access all the PlayBuffer functions: this is called a 'namespace' and it's just a useful way of organising projects in C++. Once the **PlayManager** has been created then you can include any code in this function that sets up the starting state for your game (loading resources, creating objects, playing music, etc.) This is useful for code which only needs to run once at the start of your application.

All **PlayBuffer** programs also need two more functions called MainGameUpdate() and MainGameExit(). The first of these (Listing 3: Lines 15-22) is where most of the action happens in your game as this function gets called once for every 'frame' of your game. Like animations, games are made up of static images which appear in quick succession to create the illusion of movement. **PlayBuffer** usually runs at 60 frames per second (as most commercial games do), so each frame lasts only 17 milliseconds! In each of those frames the MainGameUpdate() function needs to begin by clearing the drawing buffer to erase the previous image (Listing 3: Line 18) and then draw the current state of the game. Here we're simply drawing the traditional "Hello World" greeting in the centre of the display (Listing 3: Line 19). Each frame concludes the drawing process by "presenting" the drawing buffer (Listing 3: Line 21), which basically just means copying the drawing buffer to the Window.

```
15: // Called by the PlayBuffer once for each frame (60 times a second!)
16: bool MainGameUpdate( float elapsedTime )
17: {
18:     Play::ClearDrawingBuffer( Play::cOrange );
19:     Play::DrawDebugText( { DISPLAY_WIDTH/2, DISPLAY_HEIGHT/2 }, "Hello World!" );
20:     Play::PresentDrawingBuffer();
21:     return Play::KeyDown( VK_ESCAPE );
22: }
```

Listing 3: *A minimal "Hello World"* ***MainGameUpdate()*** *function for* ***PlayManager****.*

Note that the MainGameUpdate() function returns a Boolean type (bool) which indicates to the **PlayBuffer** whether the player has chosen to quit the game (3: 16). Here we're returning the status of the Play::KeyDown() function, passing through the 'escape' virtual key code (3: 21). So, if the escape key is pressed, then the function returns true and the game will quit.

The MainGameExit() function (4: 25-29) is a little less exciting and is called just once as your game application exits. It's important that this function frees up any memory allocated by your program, but as that's typically all handled in **PlayManager** for you, you probably just need to call Play::DestroyManager() as in the example below. Finally, it returns PLAY_OK to indicate that the program finished without any errors.

```
24: // Gets called once when the player quits the game
25: int MainGameExit( void )
26: {
27:     Play::DestroyManager();
28:     Return PLAY_OK;
29: }
```

Listing 4: *The **MainGameExit()** function for **PlayManager**.*

NEXT STEPS

Thirty lines of code to write "Hello World" might seem like a lot, but it's not hard to move from this simple example to a fully interactive game! Over the coming pages we'll do exactly that by showing you how to change the "Hello World" example into a simple shoot-'em-up game. We'll do this in a number of stages, but it is really important to note:

In each stage we will only highlight the lines of code you need to add or change from the previous stage. If a function doesn't change at all between stages then it won't appear in the new listing, but that doesn't mean you should delete it.

Also, if you see any code with a blue background like this, then it's just there for reference and you're **NOT** expected to type it in!

Stage 1: Let There be Sprites!

The first stage in our tutorial game will be to display some sprites on the screen, but before we do that, take a look in your Data\Sprites directory inside your **Visual Studio** project folder. Here you should see a range of .png format graphics files which we'll use to create our game. We use this particular file format because it supports transparency (pixels you can fully or partially see-through) and creates small file sizes without losing any image quality.

```
#define PLAY_IMPLEMENTATION
#define PLAY_USING_GAMEOBJECT_MANAGER
#include "Play.h"

int DISPLAY_WIDTH = 1280;
int DISPLAY_HEIGHT = 720;
int DISPLAY_SCALE = 1;

struct GameState
{
    float timer = 0;
    int spriteId = 0;
};

GameState gameState;

// The entry point for a PlayBuffer program
void MainGameEntry( PLAY_IGNORE_COMMAND_LINE )
{
       Play::CreateManager( DISPLAY_WIDTH, DISPLAY_HEIGHT, DISPLAY_SCALE );
}

// Called by the PlayBuffer once every frame (60 times a second!)
bool MainGameUpdate( float elapsedTime )
{
    gameState.timer += elapsedTime;
    Play::ClearDrawingBuffer( Play::cOrange );

    Play::DrawDebugText( { DISPLAY_WIDTH / 2, DISPLAY_HEIGHT/2 },
                         Play::GetSpriteName( gameState.spriteId ),
                         Play::cWhite );

    Play::DrawSprite( gameState.spriteId, Play::GetMousePos(), gameState.timer );

    if( Play::KeyPressed( VK_SPACE ) )
        gameState.spriteId++;

    Play::PresentDrawingBuffer();
    return Play::KeyDown( VK_ESCAPE );
}

// Called once by the PlayBuffer when the application quits
void MainGameExit( void )
{
    Play::DestroyManager();
}
```

***Listing 5:** Modifications to the program to allow sprites to be previewed.*

The **PlayManager** will automatically load all the .png files in this directory and turn them into 'sprites'. 'Sprites are just a game programming term for an image which moves around the screen. Note that some of the files have an underscore towards the end of their filenames which indicates that they are animated sprites with multiple frames of animation. For example, agent8_climb_4.png contains 4 frames of animation (see figure 4).

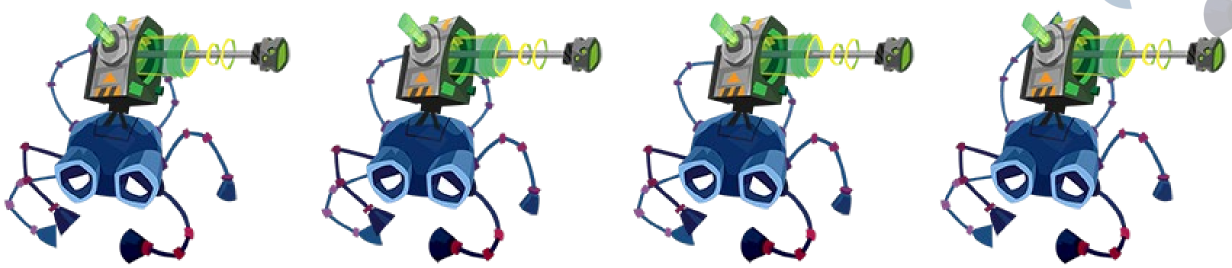


Figure 4: *An animated sprite image*

Now modify the code as shown in listing 5. Any code shown in light orange doesn't need to be changed but helps to give the surrounding context. These changes include increasing the size and resolution of the window to something less retro (5: 5-7) and creating a brand new GameState structure (5: 9-13). We're going to maintain this struct throughout this tutorial to "remember" the overall state of our game, but to begin with we'll just use it to store a timer and an id for the 'current' sprite. Note that we create an instance of GameState in global scope before MainGameEntry() (5: 15) so that the game's state can be accessed from anywhere in our code.

Next, we add elapsedTime to the timer variable in GameState (5: 26). The elapsedTime variable is passed through to the MainGameUpdate() function and contains the number of seconds that have passed since the last frame (usually about 0.017). Adding this to the timer gives us a running total of the number of seconds that have passed since our application started, but more importantly in this example, it gives us a way of driving the animation of a sprite.

Change the call to DrawDebugText() so that it draws the name of the currently selected sprite (5: 29-31). The Play::GetSpriteName() function provides the text for a sprite's name based on its unique 'id' number. These ids are automatically assigned to each sprite as they are loaded, starting from 0 and working upwards. Our gameState.spriteId variable is initialised to 0 so using this as an id will provide the name of the very first sprite. It's fine to use the result of one function as an argument to another like this, provided the types match up. Here Play::GetSpriteName() returns a char* type and the Play::DrawDebugText() function takes a char* type as its second argument, so you can use the function call in place of the argument.

We use the spriteId variable in a similar way to draw the sprite (5: 33), passing through the position of the mouse (another function call as an argument), and the GameState timer, to provide the index of the animation frame. It doesn't matter that the timer's value will exceed the number of frames in any animation as the drawing function will automatically wrap around back to the start of the animation again.

Finally, we use the '++' operator to add one to the spriteId when the space bar is pressed, so we can see the next sprite (5: 35-36). Note that we've not used curly brackets around the code which depends on this 'if' condition. This is legal in C++ if it is just for a single line of code like this, but we'd recommend you always use curly brackets as it's not "good practice". We're only doing it because it helps us to save space on the page when printing these tutorials.

If you run the application now, then you should be able to preview all the different sprites in the Data\Sprites directory. Take a look through all the sprites now. Note that we have font sprites where each character in the font is a different frame of the animation! You should also find that the application aborts with an error when you reach the end of all the sprites. That's because spriteId is now bigger than the total number of sprites loaded by the system! Press the stop button on the taskbar to end the application.

A "PAINTER'S" ALGORITHM

Before proceeding too much further, it's worth understanding how things get drawn to the window using PlayBuffer. There is an important distinction between the 'drawing buffer' which acts as a canvas for building up your game image, and the game window which appears on your desktop. All drawing functions in **PlayBuffer** draw to an invisible 'drawing buffer' to hide the drawing process. Each frame of the game is constructed by drawing background images first and successively drawing images on top. The order in which you draw your sprites controls which appear in front of others. This approach is called a "painter's algorithm" as it mirrors the way in which traditional artists paint their scenes: starting with the background objects and moving towards the foreground (see figure 5: note that clearing the drawing buffer is unnecessary if you are drawing a full-screen background image). However, this drawing process is invisible to the player, as they see the image from the previous frame's in the application window. When drawing is complete, Play::PresentDrawingBuffer() updates the visible window with the contents of the drawing buffer. It might help you understand this process if you try commenting out the call to Play::ClearDrawingBuffer() in the example above (5: 27) to see what happens.

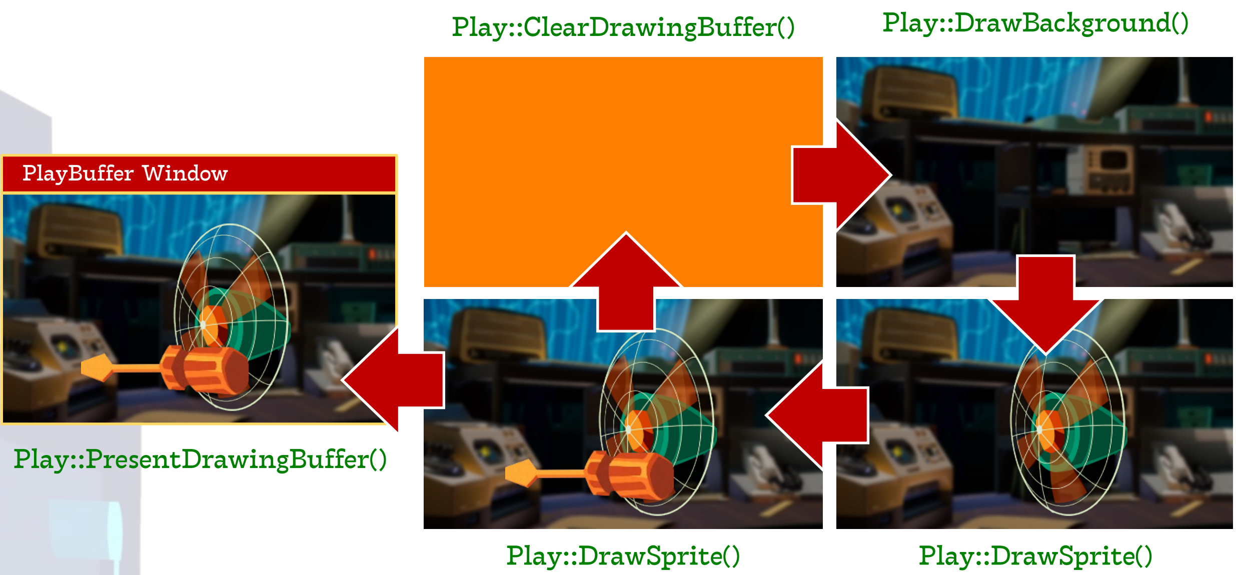


Figure 5: *Each frame, the background is drawn to the drawing buffer first, and each draw operation appears on top of any previous ones. Only when the PresentDrawingBuffer() function is called, does the drawing buffer get copied to the window so the player can see it.*

CO-ORDINATES, AXES AND ORIGINS

There are a couple more key concepts that you should be aware of when using the **PlayBuffer**. Most co-ordinate systems on computer displays work differently to co-ordinates in school maths. In school we typically draw the origin (coordinate x=0, y=0) in the bottom, left, with the y-axis increasing up the page, and the x-axis increasing to the right. However, computer displays generally place the origin in the **top**, **left** with the y-axis increasing **down** the screen and the x-axis increasing to the right (see figure 6).

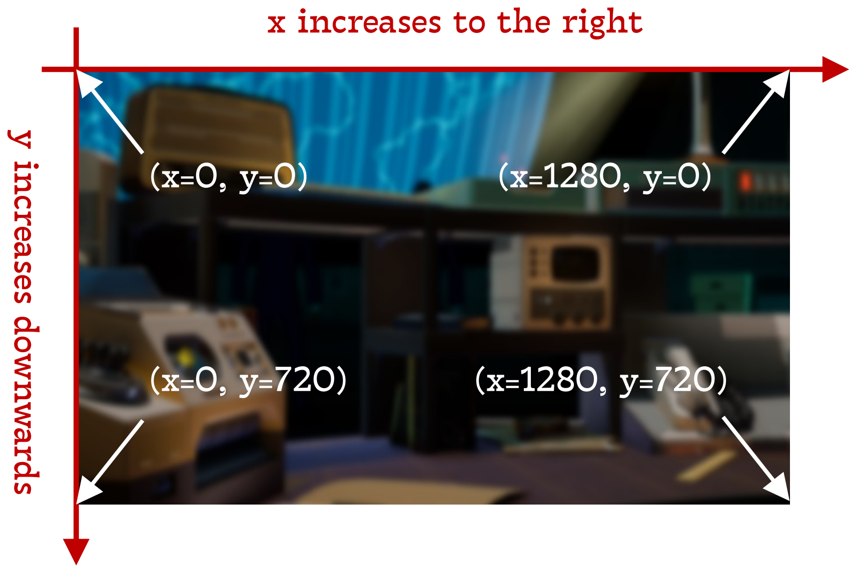


Figure 6: *The co-ordinate system used in* ***PlayBuffer***

By default, all sprites also have their own local 'origins' set to the top left corner of the sprite. The sprite origin controls the position from which the sprite is manipulated in the game (including its centre of rotation). Run the program again and notice how your mouse pointer 'holds' each sprite from its top left corner (even if that 'corner' is invisible!) Now call the function, Play::CentreAllSpriteOrigins() in the MainGameEntry() function directly after the call to Play::CreateManager() and run the program again. This resets the origin of each sprite so you 'hold' each sprite from its centre instead – they would also rotate around their centres and their radial collisions would be detected relative to their centres too.

Figure 7: *Sprite origins default to the top left corner of the sprite*

Game Objects

A GameObject is a structure (or 'struct') provided by **PlayManager** for representing interactive objects in a typical game. Every GameObject has a set of common properties and **PlayManager** has a range of useful functions for managing them. The GameObject struct has already been created for you in "Play.h", but it's useful to examine it more closely:

```
struct GameObject
{
    int type;
    int spriteId;
    Point2D pos;
    Point2D oldPos;
    Vector2D velocity;
    Vector2D acceleration;
    float rotation;
    float rotSpeed;
    float oldRot;
    int frame;
    float framePos;
    float animSpeed;
    int radius;
    float scale;
};
```

We will learn about most of these data members as we go along, but it's important to note that every object must have a 'type' which defines the 'group behaviour' of that kind of GameObject. Typically we use an enum to define all the different GameObject types in a game and we'll learn more about how that works in the next stage of the example program.

```
struct GameState
{
    int score = 0;
};

GameState gameState;

enum GameObjectType
{
    TYPE_NULL = -1,
    TYPE_AGENT8,
};

void HandlePlayerControls();

// The entry point for a PlayBuffer program
void MainGameEntry( PLAY_IGNORE_COMMAND_LINE )
{
    Play::CreateManager( DISPLAY_WIDTH, DISPLAY_HEIGHT, DISPLAY_SCALE );
    Play::CentreAllSpriteOrigins();
    Play::LoadBackground( "Data\\Backgrounds\\background.png" );
    Play::StartAudioLoop( "music" );
    Play::CreateGameObject( TYPE_AGENT8, { 115, 0 }, 50, "agent8" );
}

// Called by the PlayBuffer once every frame (60 times a second!)
bool MainGameUpdate( float elapsedTime )
{
    Play::DrawBackground();
    HandlePlayerControls();
    Play::PresentDrawingBuffer();
    return Play::KeyDown( VK_ESCAPE );
}

```

```
void HandlePlayerControls()
{
    GameObject& obj_agent8 = Play::GetGameObjectByType( TYPE_AGENT8 );
    if( Play::KeyDown( VK_UP ) )
    {
        obj_agent8.velocity = { 0, -4 };
        Play::SetSprite( obj_agent8, "agent8_climb", 0.25f );
    }
    else if( Play::KeyDown( VK_DOWN ) )
    {
        obj_agent8.acceleration = { 0, 1 };
        Play::SetSprite( obj_agent8, "agent8_fall", 0 );
    }
    else
    {
        Play::SetSprite( obj_agent8, "agent8_hang", 0.02f );
        obj_agent8.velocity *= 0.5f;
        obj_agent8.acceleration = { 0, 0 };
    }
    Play::UpdateGameObject( obj_agent8 );

    if( Play::IsLeavingDisplayArea( obj_agent8 ) )
        obj_agent8.pos = obj_agent8.oldPos;

    Play::DrawLine( { obj_agent8.pos.x, 0 }, obj_agent8.pos, Play::cWhite );
    Play::DrawObjectRotated( obj_agent8 );
}
```

Listing 6: *Modifications to the program to introduce a controllable "Agent 8"*

Stage 2: Player Controls

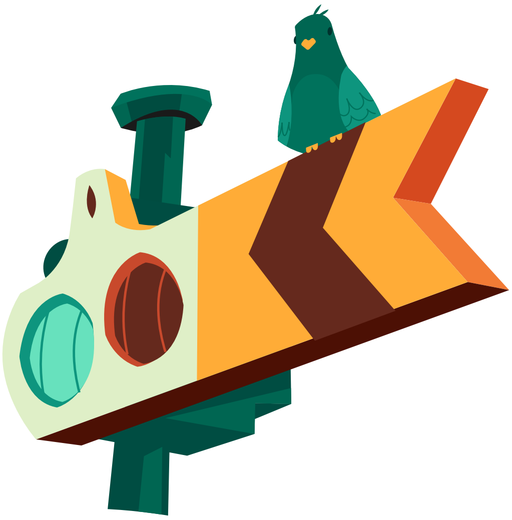
The next stage in our tutorial game will be to create the player's avatar and set up keyboard controls for it. Agent 8 is a robot spider, so we're going to have him dangling from a thread on the left-hand side of the screen, while objects are thrown towards him from the right.

The **PlayManager** is going to handle the animation of our sprites for us, so the first thing we'll do is remove the existing member variables from the GameState structure and replace them with a score, which we are going to need later (6: 11). The next change is to introduce an enumeration to represent the GameObject types in our game (6: 16-20). Note that the **PlayManager** uses -1 to represent uninitialised game objects (typically indicating some kind of problem) and the enumeration will automatically assign the next numerical value (i.e., 0) to our first GameObjectType, "TYPE_AGENT8".

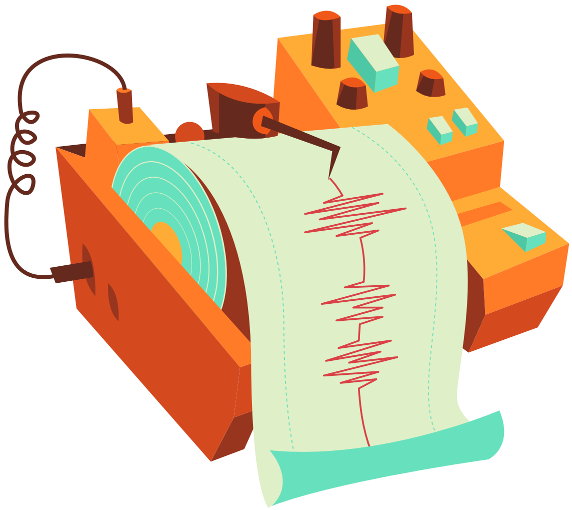
We're going to create a new function to handle the player controls, and we begin by declaring it at the start of our program (6: 22). It's necessary to 'declare' the functions in this way when we want to refer to them before we have 'defined' the code which determines how they actually work. We want to refer to this function in MainGameUpdate() (6: 38) which appears before the function is defined (6: 50-76), so we need to make this declaration.

Next, we add 4 new lines of code to MainGameEntry() (6: 28-31). You may have already included the first of these to centre all the sprite origins. Most games will benefit from sprites with origins in their centre, so it usually makes sense to do this at the start of your games. Play::LoadBackground() (6: 29) loads a large .png image as the game's main background. Note that it's necessary to specify the complete directory path and file extension for this function, and that every backslash ('\') must be written as a double backslash ('\\'). This is because the backslash symbol is used within text strings to represent special characters which aren't printable, for example, '\n' represents a new line character. Putting it twice indicates that you really do want the backslash character and not one of these special characters.

Play::StartAudioLoop() starts some dramatic music for our game (6: 30). It automatically knows to look for this in the 'Data\\Audio' directory and will play the first .mp3 file it finds with the word 'music' in its name. That works fine here, but you might need to be more careful if you have multiple music files! Finally, we call Play::CreateGameObject() using TYPE_AGENT8 as the first argument (6: 31) to create our Agent 8 object. The second argument passed to this function sets its initial position to be at the top of the screen on the left-hand side ({ 115, 0 }). The next argument is the object's collision radius in pixels (we'll come back to collisions later). The final argument tells **PlayManager** which sprite to use for this object, and here it will assign the first sprite it finds with 'agent8' in its name.



Just two lines of code need adding to MainGameUpdate(), although note that some have been deleted too (6: 37-38). Play::DrawBackground() (6: 37) now replaces Play::ClearDrawingBuffer(), as both completely reset the display buffer. Here we also call our new HandlePlayerControls() function (which we've yet to write!) (6: 38).



A NEW FUNCTION

Now we come to writing the function 'definition' for HandlePlayerControls(), which implements the desired behaviour in code. Like most of the update functions you will write in this program, we begin by retrieving a reference to the player's GameObject using Play::GetGameObjectByType() (6: 52). You should always get access your objects through the **PlayManager** using functions like this one, and never try store a global reference to a GameObject, as it might get destroyed by another part of the code in-between frames. However, now we have our reference we can access any of the GameObject's data members directly to change its behaviour.

A simple example of this can be seen in the next few lines of code (6: 53-57). This checks to see if the 'up' arrow key is held down and sets Agent8's velocity in the y-axis to -4 (upwards) when it is (6: 55). We also use the Play::SetSprite() function to change the object's sprite to the climbing animation and set the animation speed to one quarter of full speed (0.25). Setting the object's velocity like this will make it instantly change direction, but a different approach is taken when the player is pressing the 'down' arrow key (6: 58-62). Here we set the object's acceleration to increase slowly in the y-axis (downwards) to give the impression of it falling under gravity. The final 'else' condition (6: 63-68) controls what happens when neither arrow key is being held. Here we slow the velocity by multiplying by a fraction (6: 66) and set the acceleration to 0 (6: 67). Note that it's perfectly fine to multiply a 2D vector by a single value like this: it just multiplies both the x and y components of the vector by the same value.

After setting up the GameObject properties, we now need to call Play::UpdateGameObject() to get **PlayManager** to apply their effects to the object (more about that below). Once these have been applied, we check to see if the object is leaving the display area and move it back to its previous position when it is (6: 71-72). This prevents the player from leaving the playing area.

Finally, we just need to draw the player object and the web. The Play::DrawLine() function draws a vertical white line from the top of the display above the player's position to the centre of the player's sprite (6: 74). We've used the Play::DrawObjectRotated() function (6: 75) rather than the faster Play::DrawObject() function as we want to rotate the player's sprite later on. Don't forget to finish off the function body with a closing curly bracket!

SPYDER

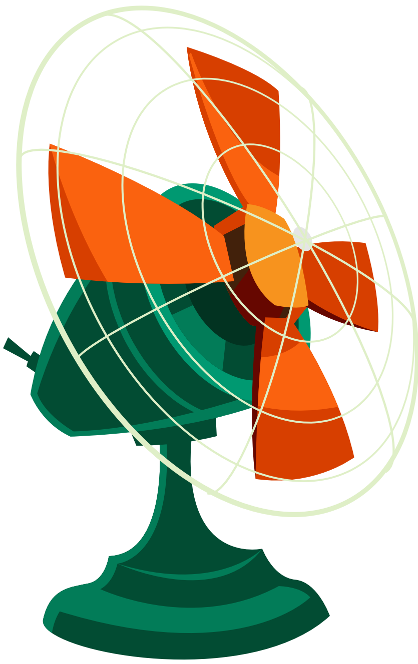
GAMEOBJECT UPDATES

If you run the program now, then you should be able to see and control Agent 8 as he climbs up and falls down the screen. The code below provides a bit more insight into what happens inside the Play::UpdateGameObject() function to make the position of the GameObject change according to its various properties. Have a scan through and see if you can work out how it works, but there's no need to type any of this as it's already implemented in "Play.h":

```
void UpdateGameObject( GameObject& obj )
{
    if( obj.type == -1 ) return; // Don't update noObject

    // Save the current position in case we need to go back
    obj.oldPos = obj.pos;
    obj.oldRot = obj.rotation;

    // Move the object according to a very simple physical model
    obj.velocity += obj.acceleration;
    obj.pos += obj.velocity;
    obj.rotation += obj.rotSpeed;

    // Handle the animation frame update
    obj.framePos += obj.animSpeed;
    if( obj.framePos > 1.0f )
    {
        obj.frame++;
        obj.framePos -= 1.0f;
    }
}
```

Stage 3: A Dangerous Fan-Base

Next, we're going to introduce a desk fan on the right-hand side of the screen which is so powerful that it has taken off and is blowing dangerous objects around (maybe this is taking place inside a spaceship, so gravity is reduced). We're going to implement this in two stages (listing 7 and listing 8) so that the code is easy to examine a page at a time!



```
enum GameObjectType
{
        TYPE_NULL = -1,
        TYPE_AGENT8,
        TYPE_FAN,
        TYPE_TOOL,
        TYPE_COIN,
        TYPE_STAR,
        TYPE_LASER,
        TYPE_DESTROYED,
};

void HandlePlayerControls();
void UpdateFan();
void UpdateTools();

// The entry point for a PlayBuffer program
void MainGameEntry( PLAY_IGNORE_COMMAND_LINE )
{
    Play::CreateManager( DISPLAY_WIDTH, DISPLAY_HEIGHT, DISPLAY_SCALE );
    Play::CentreAllSpriteOrigins();
    Play::LoadBackground( "Data\\Backgrounds\\spr_background.png" );
    Play::StartAudioLoop( "music" );
    Play::CreateGameObject( TYPE_AGENT8, { 115, 0 }, 50, "agent8" );
    int id_fan = Play::CreateGameObject( TYPE_FAN, { 1140, 217 }, 0, "fan" );
    Play::GetGameObject( id_fan ).velocity = { 0, 3 };
    Play::GetGameObject( id_fan ).animSpeed = 1.0f;
}

// Called by the PlayBuffer once every frame (60 times a second!)
bool MainGameUpdate( float elapsedTime )
{
    Play::DrawBackground();
    HandlePlayerControls();
    UpdateFan();
    UpdateTools();
    Play::PresentDrawingBuffer();
    return Play::KeyDown( VK_ESCAPE );
}
```

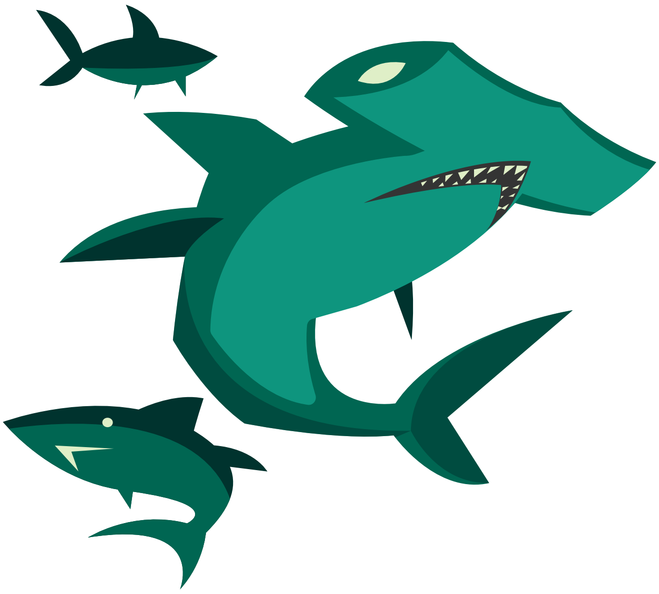
```
void UpdateFan()
{
    GameObject& obj_fan = Play::GetGameObjectByType( TYPE_FAN );
    Play::DrawObject( obj_fan );
}

void UpdateTools()
{
    GameObject& obj_agent8 = Play::GetGameObjectByType( TYPE_AGENT8 );
    std::vector<int> vTools = Play::CollectGameObjectIDsByType( TYPE_TOOL );

    for( int id : vTools )
    {
        GameObject& obj_tool = Play::GetGameObject( id );
    }
}
```

***Listing 7:** Modifications to the program to make preparations for enemy objects.*

We begin by adding all the different types we're going to need for our complete game to the enumeration (7: 20-25). We're only dealing with fans and tools in this section, but we may as well complete the list now so we don't have to come back to it later. We're then 'declaring' two new functions called UpdateFan() and UpdateTools(), alongside our first (7: 29-30) and calling them in MainGameUpdate() (7: 50-51).

Like the player's GameObject, we want the fan to exist right from the start of the game, so we're creating it in MainGameEntry() on the far right-hand side of the display (7: 40). This time we also want to start the fan moving and animating, so we're using the object's id returned from Play::CreateGameObject() (7: 40) as a parameter to Play::GetGameObject() to get a reference to the GameObject. Notice how we're treating the function Play::GetGameObject() as if it is a GameObject reference and accessing its data members directly from the function using .velocity and .animSpeed (7: 41-42). This is allowed because Play::GetGameObject() returns a GameObject reference and it saves us having to create a new GameObject reference variable in this case.

Finally, we create skeleton functions for our enemy updates (7: 91-106). The UpdateFan() function mirrors the first and last lines of our HandlePlayerControls() function, but obviously getting and drawing the fan instead. We don't want to ever rotate our fan, so we just use the faster, Play::DrawObject() function. The UpdateTools() function is a bit more interesting. We're getting a reference to Agent 8 as Tools will need to detect collisions with the player, but as there will be more than one tool, we'll need to handle them slightly differently.



A DIFFERENT SORT OF VECTOR

Our program already uses a Vector2D struct to represent vector values, and this is exactly the kind of vector you're probably used to using. It has an x and a y component and can be used to represent things like velocity which have a magnitude and a direction. However, C++ has another kind of 'vector' which means something completely different. A std::vector is a kind of container which can store sequences of other data types. We declare the vTools vector in UpdateTools() to create a local vector of integer values in which to store the ids of all the tool objects (7: 100). The **PlayManager** function Play::CollectGameObjectIDsByType() finds all the ids of objects with the given type and returns them in a vector of integers, copying them over to our local vector.

Using a std::vector, we can create a for loop which goes through each item in the vector in turn and assigns their unique id to the local id variable (7: 102). That id can then be used to retrieve a reference to the corresponding GameObject so that each object can be updated in turn. This approach provides a neat way of processing sequences of objects which can be any size. You should now be able to run the program to check it compiles and that the fan appears, but there are no other visible changes at this stage.

Stage 4: Updating the Odds

Next, we're going to implement the game logic inside these update functions. The behaviour that we're aiming for from the fan object is to move up and down the screen, randomly spawning tool objects. We achieve this using the Play::RandomRoll() function, where the integer parameter represents the number of sides on a die. So, the test that we're making at the start of UpdateFan() can be thought of as rolling 50 on a 50-sided die (8: 94). You might think that's not going to happen very often, but remember that we're rolling this die sixty times a second, so it's going to happen more than one per second on average!

Next, we create the tool object at the fan's position and randomly determine its velocity (8: 96-98). To vary the tool's direction, we're using Play::RandomRollRange() to get a random integer between -1 and 1 (i.e., -1, 0 or 1) and multiplying it by 6 to produce the y-axis velocity for the tool. Depending on the roll it will move diagonally up, straight across, or diagonally down, adding a bit of variety to the gameplay.

The next condition (8: 100) gives the tool a 50/50 chance of becoming a spanner tool instead of a screwdriver. If the roll is successful then the tool's sprite, velocity, collision radius and rotation speed are all changed. Objects don't rotate by default but setting the rotSpeed makes an object rotate automatically on each call of UpdateGameObject(). Finally (within the condition to create a new tool) we play a spawning sound effect (8: 107).

We call Play::UpdateGameObject() next to process the object's movement (8: 109). Once its position has been updated, we can check to see if the fan is leaving the display area, resetting its position and reversing its y-axis velocity if it is (8: 111-115). We are already drawing the fan object at the end (8: 116), so we're done on this update function.

```
void UpdateFan( GameState& state )
{
    GameObject& obj_fan = Play::GetGameObjectByType( TYPE_FAN );
    if( Play::RandomRoll( 50 ) == 50 )
    {
        int id = Play::CreateGameObject( TYPE_TOOL, obj_fan.pos, 50, "driver" );
        GameObject& obj_tool = Play::GetGameObject( id );
        obj_tool.velocity = Point2f( -8, Play::RandomRollRange( -1, 1 ) * 6 );

        if( Play::RandomRoll( 2 ) == 1 )
        {
            Play::SetSprite( obj_tool, "spanner", 0 );
            obj_tool.radius = 100;
            obj_tool.velocity.x = -4;
            obj_tool.rotSpeed = 0.1f;
        }
        Play::PlayAudio( "tool" );
    }
    Play::UpdateGameObject( obj_fan );

    if( Play::IsLeavingDisplayArea( obj_fan ) )
    {
        obj_fan.pos = obj_fan.oldPos;
        obj_fan.velocity.y *= -1;
    }
    Play::DrawObject( obj_fan );
}

void UpdateTools( GameState& state )
{
    GameObject& obj_agent8 = Play::GetGameObjectByType( TYPE_AGENT8 );
    std::vector<int> vTools = Play::CollectGameObjectIDsByType( TYPE_TOOL );

    for( int id : vTools )
    {
        GameObject& obj_tool = Play::GetGameObject( id );

        if( Play::IsColliding( obj_tool, obj_agent8 ) )
        {
            Play::StopAudioLoop( "music" );
            Play::PlayAudio( "die" );
            obj_agent8.pos = { -100, -100 };
        }
        Play::UpdateGameObject( obj_tool );

        if( Play::IsLeavingDisplayArea( obj_tool , Play::VERTICAL ) )
        {
            obj_tool.pos = obj_tool.oldPos;
            obj_tool.velocity.y *= -1;
        }
        Play::DrawObjectRotated( obj_tool );

        if( !Play::IsVisible( obj_tool ) )
            Play::DestroyGameObject( id );
    }
}
```

Listing 8: Modifications to the program to add the enemy objects.

TOOL UPDATE

Tool objects need to be deadly to Agent 8, so we begin their update logic with a collision test using Play::IsColliding() (8: 128-133). When colliding, we stop the music and play a death sound effect instead (8: 130-131), before hiding the player off screen (8: 132). It doesn't make sense to destroy the player as it would need to be created again to restart the game anyway!

The next bit of code (8: 134-141) should be familiar by now. There are a lot of similar things that need to happen in object update functions, and hopefully you're starting to see the patterns! Nonetheless, the last two lines are new, and check to see if the tool is visible to the game (8: 143) and destroy it if it has left the playing area completely (8: 144). Note that the exclamation mark '!' in front of the call to Play::IsVisible() reverses the logic, so it means if the player is ***not*** visible then destroy it. It is important to make sure that you destroy objects like this when you are continuously creating new ones, otherwise your game will eventually slow to a halt with all the memory and processing power required to manage them.

You should now have something which compiles and runs and looks a bit more like a game. You'll need to dodge the tools to stay alive, but you can't collect anything or retaliate yet.

Stage 5: Adding Shiny Stuff

To balance the onslaught of endless flying tools, we're going to add some bonuses for the player to collect as well. They'll be rewarded for collecting them and we'll reinforce that behaviour with a particle effect. We happen to have some coin graphics handy, so we'll use those!

Before we write the update function for these GameObjects, we'll declare it before MainGameEntry() (9: 31) and call it in MainGameUpdate() (9: 53) like we've done several times already. Coins get created randomly by the fan object in its update function (9: 111-117) in a way which you should also be familiar with by now. A larger number of 'sides' on our die (150) means they appear less often than the tools.

The UpdateCoinsAndStars() function itself is the most complicated so far, but most of it should still be familiar (10: 156-201). However, it's important to note the way in which we are handling the coins being destroyed when they are collected.

```
void HandlePlayerControls();
void UpdateFan();
void UpdateTools();
void UpdateCoinsAndStars();
```

```
bool MainGameUpdate( float elapsedTime )
{
    Play::DrawBackground();
    HandlePlayerControls();
    UpdateFan();
    UpdateTools();
    UpdateCoinsAndStars();
    Play::PresentDrawingBuffer();
    return Play::KeyDown( VK_ESCAPE );
}
```

```
void UpdateFan()
{
    GameObject& obj_fan = Play::GetGameObjectByType( TYPE_FAN );
    if( Play::RandomRoll( 50 ) == 50 )
    {
        int id = Play::CreateGameObject( TYPE_TOOL, obj_fan.pos, 50, "driver" );
        GameObject& obj_tool = Play::GetGameObject( id );
        obj_tool.velocity = Point2f( -8, Play::RandomRollRange( -1, 1 ) * 6 );

        if( Play::RandomRoll( 2 ) == 1 )
        {
            Play::SetSprite( obj_tool, "spanner", 0 );
            obj_tool.radius = 100;
            obj_tool.velocity.x = -4;
            obj_tool.rotSpeed = 0.1f;
        }
        Play::PlayAudio( "tool" );
    }
    if( Play::RandomRoll( 150 ) == 1 )
    {
        int id = Play::CreateGameObject( TYPE_COIN, obj_fan.pos, 40, "coin" );
        GameObject& obj_coin = Play::GetGameObject( id );
        obj_coin.velocity = { -3, 0 };
        obj_coin.rotSpeed = 0.1f;
    }
    Play::UpdateGameObject( obj_fan );

    if( Play::IsLeavingDisplayArea( obj_fan ) )
    {
        obj_fan.pos = obj_fan.oldPos;
        obj_fan.velocity.y *= -1;
    }
    Play::DrawObject( obj_fan );
}
```

Listing 9: *Modifications to the program to prepare for coins and stars.*

```
157: void UpdateCoinsAndStars()
158: {
159:     GameObject& obj_agent8 = Play::GetGameObjectByType( TYPE_AGENT8 );
160:     std::vector<int> vCoins = Play::CollectGameObjectIDsByType( TYPE_COIN );
161: 
162:     for( int id_coin : vCoins )
163:     {
164:         GameObject& obj_coin = Play::GetGameObject( id_coin );
165:         bool hasCollided = false;
166: 
167:         if( Play::IsColliding( obj_coin, obj_agent8 ) )
168:         {
169:             for( float rad{ 0.25f }; rad < 2.0f; rad += 0.5f )
170:             {
171:                 int id = Play::CreateGameObject(TYPE_STAR,obj_agent8.pos,0,"star");
172:                 GameObject& obj_star = Play::GetGameObject( id );
173:                 obj_star.rotSpeed = 0.1f;
174:                 obj_star.acceleration = { 0.0f, 0.5f };
175:                 Play::SetGameObjectDirection( obj_star, 16, rad * PLAY_PI );
176:             }
177: 
178:             hasCollided = true;
179:             gameState.score += 500;
180:             Play::PlayAudio( "collect" );
181:         }
182: 
183:         Play::UpdateGameObject( obj_coin );
184:         Play::DrawObjectRotated( obj_coin );
185: 
186:         if( !Play::IsVisible( obj_coin ) || hasCollided )
187:             Play::DestroyGameObject( id_coin );
188:     }
189: 
190:     std::vector<int> vStars = Play::CollectGameObjectIDsByType( TYPE_STAR );
191: 
192:     for( int id_star : vStars )
193:     {
194:         GameObject& obj_star = Play::GetGameObject( id_star );
195: 
196:         Play::UpdateGameObject( obj_star );
197:         Play::DrawObjectRotated( obj_star );
198: 
199:         if( !Play::IsVisible( obj_star ) )
200:             Play::DestroyGameObject( id_star );
201:     }
202: }
```

Listing 10: *Modifications to the program to implement coins and stars.*

Instead of getting rid of coins as soon as we detect a collision with the player, we are using a hasCollided variable (10: 178) to record that this has happened, and then destroying it later when we test to see if it is no longer visible (10: 186-187). Note the use of the OR operator (two vertical lines: '||'), along with the NOT operator (exclamation mark: '!') to create the condition, "if NOT visible OR has collided" (10: 186). We've done this because calling DestroyGameObject() at the wrong time can create unintended problems. Imagine we destroyed the object inside the if statement that detects the collision (e.g., 10: 178). That would mean that obj_coin would no-longer be a valid object reference when it is accessed again later (10: 183-184) and the game would crash! So – in general – if you are going to destroy an object then you need to do it at the last point in the code where you are working on that object. So long as it is destroyed last then nothing should accidently use it again afterwards!

In this section of code, we're also creating some star 'particles' (10: 169-176). There's nothing particularly new here, but it's worth noting that programming languages tend to measure angles in radians rather than degrees. As such we've set up a for loop which starts at 0.25 and then adds 0.5 in each iteration until it's completed a full circle. When we multiply these values by Pi we will get values in radians that produce 4 stars in diagonally opposite corners from the coin (see figure 8). The stars have a very simple update loop, so we've included that in the same function below the coin update function that creates them (10: 191-201).

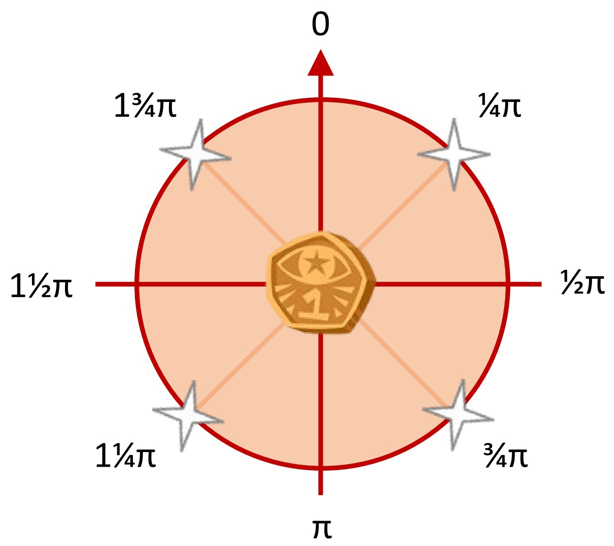


Figure 8: Angles are measured from 0 to 2π radians

Stage 6: Laser Show!

Okay, so it's about time Agent 8 defended himself, so let's implement lasers next. We're also going to create a 'destroyed' object, for when targets are hit by lasers. We want things that are hit by lasers to gradually fade out rather than instantly disappear, and we'll achieve that by changing their type into a 'destroyed' object.

We begin by declaring (11: 32-33) and then calling (11: 56-57) the two new update functions. Next, we add code to HandlePlayerControls() which creates the lasers when the player presses the space bar (11: 88-94). Note the use of Play::KeyPressed() rather than Play::KeyDown() as we only want a single laser to be created each time the player presses a key rather than constantly while the key is held down. Also note the use of a Vector2D to offset the starting position of the laser. This is necessary because the end of Agent 8's laser gun is 155 pixels to the right and 75 pixels up, relative to the player's sprite's origin (see figure 9).

The UpdateLasers() function (12: 215-256) is fairly standard, and comparable to ones we've written before. We're using a hasCollided variable again (12: 230+241) to destroy the laser at the end of the loop (12: 253-254). Also note how we're changing the type of the colliding tools to that of a 'destroyed' object (12: 231+242). What's interesting about this approach, is that it doesn't change anything else about them (their sprite, velocity, etc.) and so they continue to look and behave as they did previously – yet they aren't treated the same in terms of collisions.

```
void HandlePlayerControls();
void UpdateFan();
void UpdateTools();
void UpdateCoinsAndStars();
void UpdateLasers();
void UpdateDestroyed();
```

```
bool MainGameUpdate( float elapsedTime )
{
    Play::DrawBackground();
    HandlePlayerControls();
    UpdateFan();
    UpdateTools();
    UpdateCoinsAndStars();
    UpdateLasers();
    UpdateDestroyed();
    Play::PresentDrawingBuffer();
    return Play::KeyDown( VK_ESCAPE );
}
```

```
void HandlePlayerControls()
{
    GameObject& obj_agent8 = Play::GetGameObjectByType( TYPE_AGENT8 );
    if( Play::KeyDown( VK_UP ) )
    {
        obj_agent8.velocity = { 0, -4 };
        Play::SetSprite( obj_agent8, "agent8_climb", 0.25f );
    }
    else if( Play::KeyDown( VK_DOWN ) )
    {
        obj_agent8.acceleration = { 0, 1 };
        Play::SetSprite( obj_agent8, "agent8_fall", 0 );
    }
    else
    {
        Play::SetSprite( obj_agent8, "agent8_hang", 0.02f );
        obj_agent8.velocity *= 0.5f;
        obj_agent8.acceleration = { 0, 0 };
    }
    if( Play::KeyPressed( VK_SPACE ) )
    {
        Vector2D firePos = obj_agent8.pos + Vector2D( 155, -75 );
        int id = Play::CreateGameObject( TYPE_LASER, firePos , 30, "laser" );
        Play::GetGameObject( id ).velocity = { 32, 0 };
        Play::PlayAudio( "shoot" );
    }
    Play::UpdateGameObject( obj_agent8 );

    if( Play::IsLeavingDisplayArea( obj_agent8 ) )
        obj_agent8.pos = obj_agent8.oldPos;

    Play::DrawLine( { obj_agent8.pos.x, 0 }, obj_agent8.pos, Play::cWhite );
    Play::DrawObjectRotated( obj_agent8 );
}
```

Listing 11: *Modifications to the program to prepare for lasers and destroyed objects.*

```
void UpdateLasers()
{
    std::vector<int> vLasers = Play::CollectGameObjectIDsByType( TYPE_LASER );
    std::vector<int> vTools = Play::CollectGameObjectIDsByType( TYPE_TOOL );
    std::vector<int> vCoins = Play::CollectGameObjectIDsByType( TYPE_COIN );

    for( int id_laser : vLasers )
    {
        GameObject& obj_laser = Play::GetGameObject( id_laser );
        bool hasCollided = false;
        for( int id_tool : vTools )
        {
            GameObject& obj_tool = Play::GetGameObject( id_tool );
            if( Play::IsColliding( obj_laser, obj_tool ) )
            {
                hasCollided = true;
                obj_tool.type = TYPE_DESTROYED;
                gameState.score += 100;
            }
        }

        for( int id_coin : vCoins )
        {
            GameObject& obj_coin = Play::GetGameObject( id_coin );
            if( Play::IsColliding( obj_laser, obj_coin ) )
            {
                hasCollided = true;
                obj_coin.type = TYPE_DESTROYED;
                Play::PlayAudio( "error" );
                gameState.score -= 300;
            }
        }
        if( gameState.score < 0 )
            gameState.score = 0;

        Play::UpdateGameObject( obj_laser );
        Play::DrawObject( obj_laser );

        if( !Play::IsVisible( obj_laser ) || hasCollided )
            Play::DestroyGameObject( id_laser );
    }
}

void UpdateDestroyed()
{
    std::vector<int> vDead = Play::CollectGameObjectIDsByType( TYPE_DESTROYED );

    for( int id_dead : vDead)
    {
        GameObject& obj_dead = Play::GetGameObject( id_dead );
        obj_dead.animSpeed = 0.2f;
        Play::UpdateGameObject( obj_dead );

        if( obj_dead.frame % 2 )
            Play::DrawObjectRotated( obj_dead, ( 10 - obj_dead.frame ) / 10.0f );

        if( !Play::IsVisible( obj_dead ) || obj_dead.frame >= 10 )
            Play::DestroyGameObject( id_dead );
    }
}
```

Listing 12: Modifications to the program to add lasers and destroyed objects.

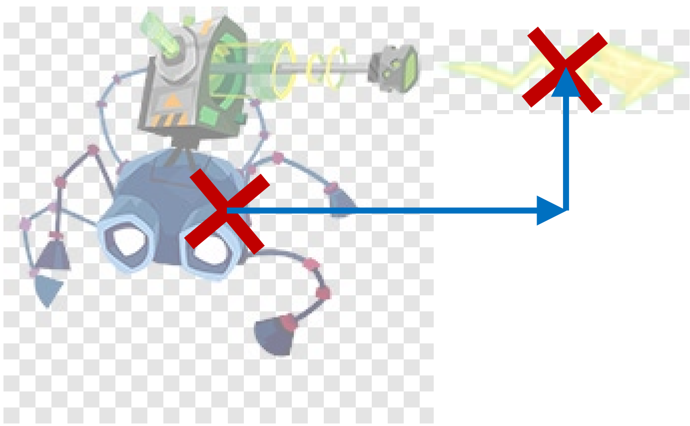


Figure 9: *The laser needs to appear 155 pixels across and 75 pixels up from Agent 8's position*

The UpdateDestroyed() function picks up control of tool and coin objects which have been hit by a laser (12: 258-274). This function doesn't do anything interesting, and that's the point: it simply allows 'destroyed' objects to continue to exist in a 'zombie' state until they finally disappear. No collision tests are being processed, so destroyed objects won't harm the player and can't be collected. The code makes use of the frame counter to work out when the object has been alive for ten animation frames and then finally destroys it (12: 271-272). However, because the animation speed is set to 5 frames per second (12: 265), this gives the object a lifetime of 2 seconds. The '%', or 'modulus' operator calculates the remainder when the first value is divided by the second, so frame % 2 will produce an answer of 1 when frame is odd and 0 when frame is even (12: 268). Conditional expressions like this will treat 0 as 'false' and any value greater than 0 as 'true', so the sprite will only get drawn on odd-numbered frames. This creates a flashing effect, and the sprite is also gradually faded out over the 10 animation frames (12: 269).

COLLISIONS

Now we have plenty of objects colliding with each other, it's probably a good time to discuss the collision radius data member of a GameObject. The best way to explain this is to simply turn on the debug mode while the game is playing by pressing the F1 key on your keyboard.

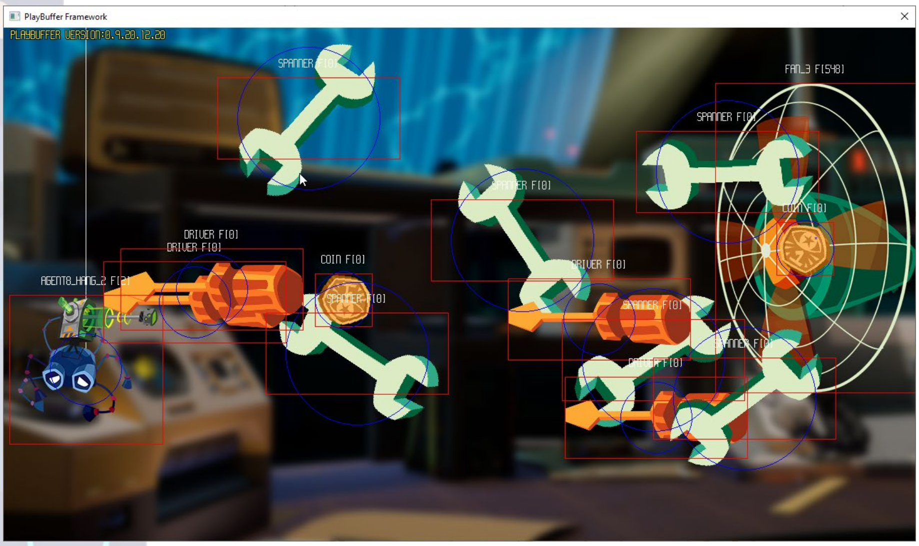


Figure 10: *Collision shapes being drawn in debug mode*

This will enable the drawing of collision shapes as well as displaying the name of each object's current sprite next to that object. The blue circles show the collision shapes used in our collision tests: when two blue circles overlap then they are treated as colliding. Changing the .radius variable changes the size of this circle and therefore the potential for collisions with this object in the game. The red rectangles show the bounding areas of the sprites themselves, but these are not used in our current collision set up.

Stage 7: The Machinery of State

Our game is now fully playable, but it doesn't start or end in a tidy way. To improve that we're going to introduce a 'state-machine' for Agent 8 which will allow us to make him behave differently at different points in the game. At the start of the game, we'd like to have Agent 8 drop in from above (STATE_APPEAR) and automatically come to a halt in the middle of the screen (STATE_HALT). At that point, 'normal' gameplay takes over (STATE_PLAY) and the player has full control of the game until Agent 8 dies (STATE_DEAD). In this state, pressing space takes us back to the starting state (STATE_APPEAR) so that the process can begin again, and we have a nice, tidy game cycle.

We will implement this using another enumeration (13: 9-15) and we'll add it as a data member of GameState (13: 20). The new UpdateAgent8() function has a familiar declaration (13: 43) and replaces the call to HandlePlayerControls() in MainGameUpdate() (13: 62). We're actually going to call HandlePlayerControls() from within the new update function, but only in STATE_PLAY. We'll make a minor tweak to Agent8's behaviour in HandlePlayerControls() by switching to STATE_HALT when he stops falling from a significant speed (13: 98-105). We have an animation of him being pulled back by his web especially for this situation (13: 101). Note that we've also deleted a few lines of code from the end of HandlePlayerControls() (13: 117+), although they will re-appear in UpdateAgent8() again in a moment!

Next, we'll take a little diversion to add some instructions to the screen using the Play::DrawFontText() function (14: 68-71). This uses a special sprite image to create text. In font sprites, each frame of the animation corresponds to a different letter or symbol in the font. You can create your own font sprites using **PlayFontTool**. The final parameter of the function determines the font justification (either Play::LEFT, Play:RIGHT or Play::CENTRE). Font sprites are normally drawn as white and can be coloured using the Play::ColourSprite() function.

```
enum Agent8State
{
    STATE_APPEAR = 0,
    STATE_HALT,
    STATE_PLAY,
    STATE_DEAD,
};

struct GameState
{
    int score = 0;
    Agent8State agentState = STATE_APPEAR;
};

void UpdateDestroyed();
void UpdateAgent8();

    Play::DrawBackground();
    UpdateAgent8(); // Replaces HandlePlayerControls() in MainGameUpdate()
    UpdateFan();

void HandlePlayerControls()
{
    GameObject& obj_agent8 = Play::GetGameObjectByType( TYPE_AGENT8 );
    if( Play::KeyDown( VK_UP ) )
    {
        obj_agent8.velocity = { 0, -4 };
        Play::SetSprite( obj_agent8, "agent8_climb", 0.25f );
    }
    else if( Play::KeyDown( VK_DOWN ) )
    {
        obj_agent8.acceleration = { 0, 1 };
        Play::SetSprite( obj_agent8, "agent8_fall", 0 );
    }
    else
    {
        if( obj_agent8.velocity.y > 5 )
        {
            gameState.agentState = STATE_HALT;
            Play::SetSprite( obj_agent8, "agent8_halt", 0.333f );
            obj_agent8.acceleration = { 0, 0 };
        }
        else
        {
            Play::SetSprite( obj_agent8, "agent8_hang", 0.02f );
            obj_agent8.velocity *= 0.5f;
            obj_agent8.acceleration = { 0, 0 };
        }
    }
    if( Play::KeyPressed( VK_SPACE ) )
    {
        Vector2D firePos = obj_agent8.pos + Vector2D( 155, -75 );
        int id = Play::CreateGameObject( TYPE_LASER, firePos , 30, "laser" );
        Play::GetGameObject( id_laser ).velocity = { 32, 0 };
        Play::PlayAudio( "SHOOT" );
    }
}
```

Listing 13: *Modifications to the program to prepare for the state-based player update*

```
    UpdateDestroyed();
    Play::DrawFontText( "64px", "ARROW KEYS TO MOVE UP AND DOWN AND SPACE TO FIRE",
        { DISPLAY_WIDTH / 2, DISPLAY_HEIGHT - 30 }, Play::CENTRE );
    Play::DrawFontText( "132px", "SCORE: " + std::to_string( gameState.score ),
        { DISPLAY_WIDTH / 2, 50 }, Play::CENTRE );
    Play::PresentDrawingBuffer();
```

```
void UpdateAgent8()
{
    GameObject& obj_agent8 = Play::GetGameObjectByType( TYPE_AGENT8 );

    switch( gameState.agentState )
    {
        case STATE_APPEAR:
            obj_agent8.velocity = { 0, 12 };
            obj_agent8.acceleration = { 0, 0.5f };
            Play::SetSprite( obj_agent8, "agent8_fall", 0 );
            obj_agent8.rotation = 0;
            if( obj_agent8.pos.y >= DISPLAY_HEIGHT / 3 )
                gameState.agentState = STATE_PLAY;
            break;

        case STATE_HALT:
            obj_agent8.velocity *= 0.9f;
            if( Play::IsAnimationComplete( obj_agent8 ) )
                gameState.agentState = STATE_PLAY;
            break;

        case STATE_PLAY:
            HandlePlayerControls();
            break;

        case STATE_DEAD:
            obj_agent8.acceleration = { -0.3f , 0.5f };
            obj_agent8.rotation += 0.25f;
            if( Play::KeyPressed( VK_SPACE ) == true )
            {
                gameState.agentState = STATE_APPEAR;
                obj_agent8.pos = { 115, 0 };
                obj_agent8.velocity = { 0, 0 };
                obj_agent8.frame = 0;
                Play::StartAudioLoop( "music" );
                gameState.score = 0;

                for( int id_obj : Play::CollectGameObjectIDsByType( TYPE_TOOL ) )
                    Play::GetGameObject( id_obj ).type = TYPE_DESTROYED;
            }
            break;

    } // End of switch on Agent8State

    Play::UpdateGameObject( obj_agent8 );

    if(Play::IsLeavingDisplayArea(obj_agent8) && gameState.agentState != STATE_DEAD)
        obj_agent8.pos = obj_agent8.oldPos;

    Play::DrawLine( { obj_agent8.pos.x, 0 }, obj_agent8.pos, Play::cWhite );
    Play::DrawObjectRotated( obj_agent8 );
}
```

Listing 14: *Modifications to the program to implement the state-based player update*

Now we can implement the final function in our example game! Note that the deleted lines from HandlePlayerControls() have reappeared at the end of UpdateAgent8() (14: 335-341). All of these things need to happen in all states – not just when the player has control of Agent8. See how we're using a switch statement to clearly separate out the states and their different behaviours. This is tidier than an equivalent set of if statements, but you must remember to finish each case with a break statement, otherwise execution just continues to the next case!

In STATE_APPEAR, we start Agent 8 moving down the screen at speed, but also introduce a downwards acceleration to represent gravity. Once Agent 8 reaches a third of the way down the screen, we switch to STATE_PLAY (14: 297-303).

You should already be familiar with how HandlePlayerControls() changes velocity and acceleration according to the keys being pressed but note how this plays out at the start of a new game. If we've just come into STATE_PLAY from STATE_APPEAR then we are already travelling downwards at speed, but (most likely) not pressing any keys. This will cause our velocity check to kick in (13: 98-103), switching to STATE_HALT and starting the halting animation.

In STATE_HALT (14: 306-310), we slow Agent 8's velocity down by multiplying by a fraction each frame, but this state only lasts for the time it takes to finish playing the animation (14: 308) and then we're back into STATE_PLAY again.

A small change to UpdateTools() will put Agent 8 into STATE_DEAD when he collides with a tool (15: 164), but we ony want him to die once, so we include a check to make sure he isn't already dead (15: 168). Back in UpdateAgent8() this will throw the player off the level with a bit of a spin (14: 317-318). More importantly though, STATE_DEAD includes code to reset the game again when the space bar is being pressed (14: 319-330). This does all the things you might expect with respect to the Agent8 GameObject, resetting its state ready for a new game and restarting the music. However, it also goes through all the tool objects destroying them so that the player doesn't start off in a difficult position (14: 328-329).

```
162:     GameObject& obj_tool = Play::GetGameObject( id );
163:
164:     if(gameState.agentState != STATE_DEAD && Play::IsColliding(obj_tool,obj_agent8))
165:     {
166:         Play::StopAudioLoop( "music" );
167:         Play::PlayAudio( "die" );
168:         gameState.agentState = STATE_DEAD;
169:     }
170:     Play::UpdateGameObject( obj_tool );
```

Listing 15: *Modifications to the UpdateTools() function to put Agent 8 into the dead state.*

Congratulations!

And that's it – you've made your first complete C++ game using the **PlayManager**! That's a great achievement, and hopefully it makes any frustration along the way with missing semi-colons and un-matched curly brackets, all seem worthwhile! We hope that this is enough to give you the C++ programming bug and that your mind is already brimming with the endless possibilities for making your own games.

PLAYMANAGER REFERENCE

Manager Setup and Shutdown

Play::CreateManager

Initialises the manager and creates a window of the required dimensions.

```
void CreateManager( int width, int height, int scale );
```

Argument	Description
width	The width of the window in pixels.
height	The height of the window in pixels.
scale	Pixel scale. One-pixel equals (scale x scale) pixels in final window.

Example:

```
constexpr int DISPLAY_WIDTH  = 1280;
constexpr int DISPLAY_HEIGHT = 720;
constexpr int DISPLAY_SCALE  = 1;

Play::CreateManager( DISPLAY_WIDTH, DISPLAY_HEIGHT, DISPLAY_SCALE );
```

Play::DestroyManager

Shuts down the manager and closes the window.

```
void DestroyManager();
```

Example:

```
Play::DestroyManager();
```

Drawing Buffer

Play::ClearDrawingBuffer

Clears the drawing buffer using the colour provided.

```
void ClearDrawingBuffer( Colour col );
```

Argument	Description
col	The colour to clear the drawing buffer with.

Example:

```
Play::ClearDrawingBuffer( Play::cOrange );
```


SPYDER

Play::LoadBackground

Loads a PNG file as a background image for the drawing buffer.

```
int LoadBackground( char* pngFilename );
```

Argument	Description
pngFilename	The full path and filename of the background image in PNG format.

Returns the index of the loaded background (indexed from zero).

Example:

```
Play::LoadBackground( "Data\\Backgrounds\\spr_background.png" );
```

Play::DrawBackground

Draws a previously loaded background image to the drawing buffer.

```
void DrawBackground( int background = 0 );
```

Argument	Description
background	Optional argument to specify the background index (defaults to zero, which is the index of the first background).

Example:

```
Play::DrawBackground();
```

Play::DrawDebugText

Draws text to the screen using the built-in debug font.

```
void DrawDebugText( Point2D pos, char* text, Colour col, bool centre );
```

Argument	Description
pos	The position to draw the text ({0,0} = top left).
text	The text you want to draw, as a string literal.
col	Optional argument to set the colour of the text (defaults to white).
centred	Optional argument to centre the text (defaults to true).

Example:

```
Play::DrawDebugText( { 50, 50 }, "Hello World!", Play::cBlue, true );
```

Play::PresentDrawingBuffer

Copies the contents of the drawing buffer to the display window. This function must be called to display the results of the drawing functions.

```
void PresentDrawingBuffer();
```

Example:

```
Play::PresentDrawingBuffer();
```

Play::GetMousePos

Gets the co-ordinates of the mouse cursor within the display buffer.

```
Point2D GetMousePos();
```

Example:

```
Point2D mousePos = Play::GetMousePos();
```

Sound Functions

Play::PlayAudio

Plays an mp3 audio file from the "Data\Sounds" directory. You should not include the file extension. Note that the sound will restart every time this function is called – this means you won't hear anything if you try to play the sound every frame!

```
void PlayAudio( char* mp3Filename );
```

Argument	Description
mp3Filename	The name of the audio file you want to play

Example:

```
Play::PlayAudio( "blast" );
```

Play::StartAudioLoop

Plays an mp3 audio file from the "Data\Sounds" directory. The audio file will loop indefinitely.

```
void StartAudioLoop ( char* mp3Filename );
```

Argument	Description
mp3Filename	The name of the audio file you want to loop

Example:

```
Play::StartAudioLoop( "music" );
```

Play::StopAudioLoop

Stops the named audio file from playing.

```
void StopAudioLoop ( char* mp3Filename );
```

Argument	Description
mp3Filename	The name of the audio file you want to stop.

Example:

```
Play::StopAudioLoop( "music" );
```

Sprite Functions

Play::GetSpriteID

Gets the sprite ID of the first matching sprite whose filename contains the given text.

```
int GetSpriteId( char* spriteName );
```

Argument	Description
spriteName	The name of the sprite you want to find the ID for.

Example:

```
int agent8_sprId = Play::GetSpriteId( "agent8" );
```

Play::GetSpriteHeight

Gets the pixel height of a sprite.

```
int GetSpriteHeight( char* spriteName );
```

Argument	Description
spriteName	The name of the sprite you want to get the height of.

Example:

```
int agent8_height = Play::GetSpriteHeight( "agent8" );
```

Play::GetSpriteWidth

Gets the pixel width of a sprite.

```
int GetSpriteWidth( char* spriteName );
```

Argument	Description
spriteName	The name of the sprite you want to get the width of.

Example:

```
int agent8_width = Play::GetSpriteWidth( "agent8" );
```

Play::GetSpriteName

Gets the stem filename for the sprite (without path or extension).

```
char* GetSpriteName( int spriteId );
```

Argument	Description
spriteId	The id of the sprite you want to find the name for.

Example:

```
const char* agent8_name = Play::GetSpriteName( agent8_ID );
```

Play::ColourSprite

Blends the sprite with the given colour (works best on white sprites). Note that colouring affects subsequent DrawSprite calls using the same sprite, but calling ColourSprite with Play::cWhite will undo any previous colouring effects.

```
void ColourSprite( char* spriteName, Colour c );
```

Argument	Description
spriteName	The name of the sprite you want to change the colour of.
c	The colour you want to blend with the sprite.

Example:

```
Play::ColourSprite( "agent8" , Play::cBlack );
```

Play::CentreSpriteOrigin

Sets the origin of the first sprite found matching the given name to the centre of the sprite (default is top left of the sprite).

```
void CentreSpriteOrigin( char* spriteName );
```

Argument	Description
spriteName	The name of the sprite you want to centre the origin of.

Example:

```
Play::CentreSpriteOrigin( "agent8_fall" );
```

Play::CentreMatchingSpriteOrigins

Centres the origins of all sprites found with matching parts of their name.

```
void CentreMatchingSpriteOrigins( char* partName );
```

Argument	Description
rootName	A common part of a sprite name to centre all the origins of.

Example:

```
Play::CentreMatchingSpriteOrigins( "agent8" );
```

Play::CentreAllSpriteOrigins

Centres the origins of all loaded sprites.

```
void CentreAllSpriteOrigins();
```

Example:

```
Play::CentreAllSpriteOrigins();
```

Play::MoveSpriteOrigin

Moves the origins of a sprite by a given number of pixels in x and y. A sprite's origin does not have to be inside the sprite itself and can be moved any distance.

```
void MoveSpriteOrigin( char* spriteName, int xoffset, int yoffset );
```

Argument	Description
spriteName	The name of the sprite you want to move the origin of.
xoffset	The number of pixels you want to move the origin to the right (negative value moves left).
yoffset	The number of pixels you want to move the origin down (negative value moves up).

Example:

```
Play:: MoveSpriteOrigin( "agent8_fall", 10, 20);
```

Play::MoveMatchingSpriteOrigins

Moves the origins of all sprites found matching the given name by a given number of pixels. A sprite's origin does not have to be inside the sprite itself and can be moved any distance.

```
void MoveMatchingSpriteOrigins(char* partName, int xoffset, int yoffset );
```

Argument	Description
rootName	The name of the sprites you want to move the origins of.
xoffset	The number of pixels you want to move the origin to the right (negative value moves left).
yoffset	The number of pixels you want to move the origin down (negative value moves up).

Example:

```
Play:: MoveMatchingSpriteOrigins( "agent8", 10, 20 );
```

Play::MoveAllSpriteOrigins

Moves the origins of all sprites by a given number of pixels in x and y. A sprite's origin does not have to be inside the sprite itself and can be moved any distance.

```
void MoveAllSpriteOrigins( int xoffset, int yoffset );
```

Argument	Description
xoffset	The number of pixels you want to move the origin to the right (negative value moves left).
yoffset	The number of pixels you want to move the origin down (negative value moves up).

Example:

```
Play::MoveAllSpriteOrigins( 10, 20 );
```

Play::DrawSprite

Draws the first matching sprite whose filename contains the given text, or draws the sprite with the its unique sprite id. Note that using a sprite id is more efficient.

```
void DrawSprite(char* spriteName, Point2D pos, int frameIndex );
```

Or

```
void DrawSprite( int spriteID, Point2D pos, int frameIndex );
```

Argument	Description
spriteName	The name of the sprite you want to draw.
spriteID	The id of the sprite you want to draw.
pos	The x/y position on the display you want to draw the sprite. Specifically, the point where the origin of the sprite will be drawn.
frameIndex	When sprites consist of multiple frames the frame index determines which frame is drawn, starting at frame 0. Where a sprite has only one frame, this argument has no effect.

Example:

```
Play::DrawSprite( "agent8", { 100, 100 }, 0 );
Play::DrawSprite( agent8_ID, { 100, 100 }, 0 );
```

Play::DrawSpriteTransparent

Draws the first matching sprite whose filename contains the given text, or the sprite with the unique sprite id, with the specified opacity. Note that this is slower than DrawSprite and so should only be used when you need transparency.

```
void DrawSpriteTransparent(char* name, Point2D pos, int frame, float opacity);
```

Or

```
void DrawSpriteTransparent( int id, Point2D pos, int frame, float opacity );
```

Argument	Description
name	The name of the sprite you want to draw.
id	The id of the sprite you want to draw.
pos	The x/y position on the display you want to draw the sprite. Specifically, the point where the origin of the sprite will be drawn.
frame	When sprites consist of multiple frames the frame index determines which frame is drawn, starting at frame 0. Where a sprite has only one frame, this argument has no effect.
opacity	Controls how transparent the sprite should be. 0 is completely transparent and 1 is fully opaque (unable to see through it at all).

Example:

```
Play:: DrawSpriteTransparent( "agent8", { 100, 100 }, 0, 0.5f );
Play:: DrawSpriteTransparent( agent8_ID, { 100, 100 }, 0, 0.5f );
```

Play::DrawSpriteRotated

Draws the first matching sprite whose filename contains the given text, or the sprite with the unique sprite id, with the specified angle, scale, and opacity. Note that this is slower than the other Draw functions and so should only be used when you need rotation or scale.

```
void DrawSpriteRotated(char* name, Point2D pos, int frame, float angle, float scale, float opacity);
```

Or

```
void DrawSpriteRotated( int id, Point2D pos, int frame, float angle, float scale, float opacity);
```

Argument	Description
name	The name of the sprite you want to draw.
id	The id of the sprite you want to draw.
pos	The x/y position on the display you want to draw the sprite. Specifically, the point where the origin of the sprite will be drawn.
frame	When sprites consist of multiple frames the frame index determines which frame is drawn, starting at frame 0. Where a sprite has only one frame, this argument has no effect.
angle	Angle in radians to rotate the sprite clockwise.
scale	Amount to scale the sprite, with 1.0f being full size, 0.5f half size, 2.0f double sized and so on.
opacity	Controls how transparent the sprite should be. 0 is completely transparent and 1 is fully opaque (defaults to 1 which means it cannot be seen through at all).

Example:

```
Play::DrawSpriteRotated( "agent8", { 100, 100 }, 0, 1.0f );
Play::DrawSpriteRotated( agent8_ID, { 100, 100 }, 0, 1.0f );
```

Play::DrawLine

Draws a single-pixel wide line between two points in the given colour.

```
void DrawLine( Point2D start, Point2D end, Colour col );
```

Argument	Description
start	The x/y coordinate for the start point of the line.
end	The x/y coordinate for the end point of the line.
c	The colour of the line.

Example:

```
Play::DrawLine ( {10, 10 }, { 40, 40 }, Play::cWhite );
```

Play::DrawCircle

Draws a single-pixel wide circle at a given origin.

```
void DrawCircle( Point2D pos, int radius, Colour col );
```

Argument	Description
pos	The x/y coordinate for the origin of the circle.
radius	The length of the circle's radius in pixels.
c	The colour of the circle.

Example:

```
Play::DrawCircle( {150, 150 }, 30, Play::cWhite );
```

Play::DrawSpriteLine

Draws a line between two points using a sprite as a 'pen', blended with the given colour. Note that colouring affects subsequent DrawSprite calls using the same sprite!

```
void DrawSpriteLine(Point2D start, Point2D end, char* sprite, Colour);
```

Argument	Description
start	The x/y coordinate for the start point of the line.
end	The x/y coordinate for the end point of the line.
sprite	Name of the sprite to be used as the 'pen' to draw the line.
col	Optional colour to blend with the sprite (defaults to white).

Example:

```
Play::DrawSpriteLine( { 10, 10 }, { 40, 40 }, "pen_8px", Play::cBlue );
```

Play::DrawSpriteCircle

Draws a circle at a given origin using a sprite as a 'pen' blended with the given colour. Note that colouring affects subsequent DrawSprite calls using the same sprite!

```
void DrawSpriteCircle( int x, int y, int radius, char* penSprite, Colour col);
```

Argument	Description
x	The x coordinate for the origin of the circle.
y	The y coordinate for the origin of the circle.
radius	The length of the circle's radius in pixels.
sprite	Name of the sprite to be used as the 'pen' to draw the circle.
col	Optional colour to blend with the sprite (defaults to white).

Example:

```
Play::DrawSpriteCircle( 75, 175, 40, "pen_8px", Play::cBlue );
```

Play::DrawFontText

Draws text using a sprite-based font exported from PlayFontTool.

```
void DrawFontText(char* fontId, std::string text, Point2D pos, Align justify);
```

Argument	Description
fontId	The unique sprite id of the font you want to use.
text	The string containing the text you want to draw.
pos	The x/y coordinate for the location for text to be drawn at.
justify	Optional argument determining whether the text is left, right, or centre justified (defaults to left justified).

Example:

```
Play::DrawFontText( spyderFontId,
                    "Level 1: Get ready!",
                    { DISPLAY_WIDTH/2, DISPLAY_HEIGHT/2 },
                    Play::CENTRE );
```

Game Objects

A GameObject is a structure (or 'struct') provided by PlayManager that represents an interactive object in a typical game. Each GameObject has a set of useful, common properties and PlayManager has a number of functions for managing them.

Properties

The following properties can be accessed and changed directly from any GameObject. Note that GameObjects should always be created by the CreateGameObject function described below otherwise some **PlayManager** GameObject functions will not behave correctly. Some **PlayManager** functions on GameObjects will change the GameObject's properties for you.

Argument	Description
type	A number representing the type of the GameObject as an int or enum. So a type of 1 might correspond to a health pickup and a type of 2 might correspond to a missile, for example. The only type value defined by PlayManager is -1, which corresponds to "no type". It is up to the user to decide how to assign other GameObject types. **PlayManager** will simply treat each unique value as a distinct type.
spriteId	The unique id of the sprite to be associated with the GameObject as used in certain DrawSprite commands.
pos	The x/y screen position where the origin of the GameObject is placed. Used for calculating collisions as well as drawing the sprite.
velocity	An x/y vector that determines how the position of the GameObject changes each time it is updated. The x-velocity will be added to the x position, and y-velocity to the y position.
acceleration	An x/y vector that determines how the velocity of the GameObject changes each time it is updated. The x-acceleration will be added to the x velocity, and y-acceleration to the y velocity.

rotation	The angle by which GameObject should be rotated when it is drawn. Measured in radians, clockwise from 12-o'clock.
rotSpeed	A constant value which is added to the rotation of the GameObject each time it is updated.
frame	The sprite frame to use when the GameObject is drawn.
animSpeed	The amount the sprite frame is increased by each time the GameObject is updated. Numbers greater than 1 will result in frame-skipping. Numbers less than 1 will slow the animation down, but are best kept to fractions that divide equally into 1 (0.2, 0.5. 0.333, etc).
radius	The distance away from GameObject's origin to detect collisions. Measured in pixels.
scale	The size to draw the sprite associated with the GameObject. 1.0f is full size, 0.5f half size, 2.0f double size, and so on.

Example:

```
int agent8_id = Play::CreateGameObject(TYPE_PLAYER, {512, 128}, 64, "agent8");
GameObject agent8 = Play::GetGameObject( agent8_id );
agent8.velocity = { 10, 0 };
agent8.pos.y = 75;
```

Play::CreateGameObject

Creates a new GameObject and adds it to the managed list. Returns the new object's unique id.

```
int CreateGameObject( int type, Point2D pos, int colRadius, char* sprName );
```

Argument	Description
type	The type of GameObject.
pos	The x/y coordinate for the origin of the GameObject.
colRadius	The distance away from GameObject's origin to detect collisions. Measured in pixels.
sprName	The name of the sprite to use for the GameObject.

Example:

```
int agent8_id = Play::CreateGameObject( TYPE_PLAYER, {512,128}, 64,"agent8" );
```

Play::GetGameObject

Retrieves a GameObject based on its id. Returns a special GameObject with a type of -1 if no object can be found with this id.

```
GameObject& GetGameObject( int id );
```

Argument	Description
id	The unique id of the GameObject you wish to retrieve.

Example:

```
GameObject agent8 = Play::GetGameObject( agent8_id );
```

Play::GetGameObjectByType

Retrieves the first GameObject matching the given type. Returns a special GameObject with a type of -1 if no object can be found with this id.

```
GameObject& GetGameObjectByType( int type );
```

Argument	Description
type	The type of the GameObject you wish to retrieve.

Example:

```
GameObject agent8 = Play::GetGameObjectByType( TYPE_PLAYER );
```

Play::CollectGameObjectIDsByType

Returns a vector containing the IDs of all of the GameObjects with the matching type. The vector will be empty if no GameObjects match the type.

```
std::vector<int> CollectGameObjectIDsByType( int type );
```

Argument	Description
type	The type of the GameObject IDs you wish to retrieve.

Example:

```
std::vector<int> enemy_IDs = Play::CollectGameObjectIDsByType( TYPE_ENEMY );
```

Play::CollectAllGameObjectIDs

Returns a vector containing the IDs of all GameObjects. The vector will be empty if there are no GameObjects.

```
std::vector<int> CollectAllGameObjectIDs();
```

Example:

```
std::vector<int> gameObject_IDs = Play::CollectAllGameObjectIDs();
```

Play::UpdateGameObject

Performs a typical update of the object's position and animation. Changes its velocity by its acceleration, its position by its velocity, its rotation by its rotation speed, and its animation frame by its animation speed.

```
void UpdateGameObject( GameObject& object );
```

Argument	Description
object	The GameObject you wish to update.

Example:

```
Play::UpdateGameObject( agent8 );
```

Play::DestroyGameObject

Deletes the GameObject with the corresponding id.

```
void DestroyGameObject( int id );
```

Argument	Description
id	The unique id of the GameObject you wish to delete.

Example:

```
Play::DestroyGameObject( agent8id );
```

Play::DestroyGameObjectsByType

Deletes all GameObjects with the corresponding type.

```
void DestroyGameObjectsByType( int type );
```

Argument	Description
type	The type of the GameObjects you wish to delete.

Example:

```
Play::DestroyGameObjectsByType( TYPE_ENEMY );
```

Play::IsColliding

Checks whether the two objects overlap based on their collision radii.

```
bool IsColliding( GameObject& obj1, GameObject& obj2 );
```

Argument	Description
obj1 obj2	The GameObjects to perform the collision check on.

Example:

```
bool shotByLaser = Play::IsColliding( agent8, laser );
```

Play::IsVisible

Checks whether any part of the object is visible within the DisplayBuffer.

```
bool IsVisible( GameObject& obj );
```

Argument	Description
obj	The GameObject to check whether it is visible.

Example:

```
bool agent8Visible = Play::IsVisible( agent8 );
```

Play::IsLeavingDisplayArea

Checks whether the object is overlapping the edge of the screen and moving outwards.

```
bool IsLeavingDisplayArea ( GameObject& obj, Direction dirn = ALL );
```

Argument	Description
obj	The GameObject to check whether it is leaving the display area.
dirn	Optional argument to restrict which axis to check. Defaults to checking both horizontal and vertical.

Example:

```
bool leaving = Play::IsLeavingDisplayArea( agent8,Play::VERTICAL );
```

Play::IsAnimationComplete

Checks whether the animation has completed playing.

```
bool IsAnimationComplete( GameObject& obj );
```

Argument	Description
obj	The GameObject to check whether its animation is complete.

Example:

```
bool agent8AnimationComplete = Play::IsAnimationComplete( agent8 );
```

Play::SetGameObjectDirection

Sets the velocity of the object based on a target rotation angle.

```
void SetGameObjectDirection( GameObject& obj, int speed, float rotation );
```

Argument	Description
obj	The GameObject you wish to change the velocity of.
speed	The number of pixels you want the GameObject to move in a straight line each time it is updated.
rotation	The angle you want the GameObject to move in.

Example:

```
Play::SetGameObjectDirection( agent8, 10, PLAY_PI);
```

Play::PointGameObject

Sets the rotation and velocity of the object to point towards the target coordinate moving at a given speed.

```
void PointGameObject( GameObject& obj, int speed, int targetX, int targetY );
```

Argument	Description
obj	The GameObject you wish to change the rotation of.
speed	The number of pixels you want the GameObject to move towards the target point in each update.
targetX	The x coordinate of the target point.
targetY	The y coordinate of the target point.

Example:

```
Play::PointGameObject ( agent8, 8, 400, 400);
```

Play::SetSprite

Changes the object's current spite and resets its animation frame to the start.

```
void SetSprite( GameObject& obj, const char* spriteName, float animSpeed );
```

Argument	Description
obj	The GameObject you wish to set the sprite for.
spriteName	The name of the sprite you wish to set for the GameObject.
animSpeed	The number of frames to increase the animation by each time the GameObject is updated.

Example:

```
Play::SetSprite( agent8, "agent8", 1 );
```

Play::DrawObject

Draws the object's sprite without rotation or transparency. This is the fastest way to draw a GameObject, and so should be the preferred method when rotation and alpha are not required.

```
void DrawObject( GameObject& obj );
```

Argument	Description
obj	The GameObject you wish draw.

Example:

```
Play::DrawObject( agent8 );
```

Play::DrawObjectTransparent

Draws the object's sprite with transparency. This is slower than DrawObject and so should only be used if you require transparency.

```
void DrawObjectTransparent( GameObject& obj, float opacity );
```

Argument	Description
obj	The GameObject you wish draw.
opacity	How transparent the sprite should be. 0.0f is fully transparent and 1.0f is fully opaque.

Example:

```
Play::DrawObjectTransparent( agent8, 0.7f );
```

Play::DrawObjectRotated

Draws the object's sprite with rotation, scale, and transparency. This is slower than DrawObject or DrawObjectTransparent and so should only be used if you require rotation or scale. Rotation and scale are taken from the properties of the GameObject.

```
void DrawObjectRotated( GameObject& obj, float opacity = 1.0f );
```

Argument	Description
obj	The GameObject you wish draw.
opacity	Optional argument determining how transparent the sprite should be. 0.0f is fully transparent and 1.0f is fully opaque. Defaults to opaque.

Example:

```
Play::DrawObjectRotated( agent8 );
```

Miscellaneous Functions

Play::KeyPressed

Returns true if the key has been pressed since it was last released. Uses Microsoft standard Virtual Key codes, a full list of which can be found here:
https://docs.microsoft.com/en-us/windows/win32/inputdev/virtual-key-codes

```
bool KeyPressed( int vKey );
```

Argument	Description
vKey	The key you wish to check.

Example:

```
bool spacePressed = Play::KeyPressed( VK_SPACE );
```

Play::KeyDown

Returns true if the key is currently being held down. Uses Microsoft standard Virtual Key codes, a full list of which can be found here:
https://docs.microsoft.com/en-us/windows/win32/inputdev/virtual-key-codes

```
bool KeyDown( int vKey );
```

Argument	Description
vKey	The key you wish to check.

Example:

```
bool spaceDown = Play::KeyDown( VK_SPACE );
```

Play::RandomRoll

Returns a random number as if you rolled a die with this many sides.

```
int RandomRoll( int sides );
```

Argument	Description
vKey	The key you wish to check.

Example:

```
int hitRoll = Play::RandomRoll( 20 );
```

Play::RandomRollRange

Returns a random number ranging from min to max inclusive.

```
int RandomRollRange( int min, int max );
```

Argument	Description
vKey	The key you wish to check.

Example:

```
int damageRoll = Play::RandomRollRange( 4, 12 );
```

LEGAL NOTICES

ACKNOWLEDGEMENTS

We would like to thank the staff and students from UTC Sheffield for trialling and testing the PlayBuffer at their 2020 after school club. We would also like to thank the students from the University of Sheffield and beyond who worked with the PlayBuffer as part of the 2021 Sheffield Women in Computer Science game development workshops.

We would like to thank the whole Spyder™ team for allowing us to use assets from their amazing game in this tutorial. If you'd like to see what Spyder's real game developers did with their game then head on over to the Apple Arcade and check it out!

www.spyderthegame.com

https://github.com/sumo-digital-academy/playbuffer