# The Hornet Engine

Contents

[The Hornet Engine 1](#_Toc187250072)

[Introduction 2](#_Toc187250073)

[Compiling the Hornet Engine 3](#_Toc187250074)

[Easy mode 3](#_Toc187250075)

[Veteran mode 3](#_Toc187250076)

[Hard mode 4](#_Toc187250077)

[A guide to the folders 6](#_Toc187250078)

[Where to program 8](#_Toc187250079)

[How to do stuff 8](#_Toc187250080)

[Result 8](#_Toc187250081)

[ErrorLogger 8](#_Toc187250082)

[Vector2D and Shapes 9](#_Toc187250083)

[HtGraphics 9](#_Toc187250084)

[HtCamera 14](#_Toc187250085)

[HtAudio 16](#_Toc187250086)

[HtKeyboard 18](#_Toc187250087)

[HtMouse 19](#_Toc187250088)

[HtGameController 21](#_Toc187250089)

[HtUnifiedInput 22](#_Toc187250090)

[HornetMenus 22](#_Toc187250091)

[HornetApp 22](#_Toc187250092)

[Settings 22](#_Toc187250093)

[Angle 23](#_Toc187250094)

[GameObject 23](#_Toc187250095)

[ObjectManager 24](#_Toc187250096)

[Event 24](#_Toc187250097)

[Memory Leaks Detected 24](#_Toc187250098)

## Introduction

The Hornet Engine is not really an engine. It is a thin wrapper to the Simple Direct Media Layer, or SDL. (It currently uses SDL2, but I will probably update to SDL3 at some point.) It is used as a teaching tool for students who are learning C++ and want to program games. It therefore has a few oddities as discussed below.

The point is that I want the students to learn C++ but also make games. To make a game, the obvious thing to do is use a game engine. The problem with most engines is that they are too good. The students won’t be really learning C++. Instead, they will be learning the engine function calls, which won’t help them as much when using other engines. SDL is a great solution. It allows the programmer to draw in 2D, get input and play sounds, while still having to write basic code. My students are just starting out, so I need an easier interface to avoid me having to explain everything to them at once.

The Hornet Engine has some strange features. There are three reasons:

1. It’s a teaching tool. The start of the course, my students don’t know C++ but I want to teach them games programming as well as OO architecture. They want to get into programming a game as quickly as possible, so I can make them wait forever while I teach them a lot of theory. The theory comes during the course, not all at the start. For most of the time they don’t know about callbacks, or pointers and are a bit vague on how two classes can interact. Some of them are Java programmers and are a bit vague about references because it all happens by magic in Java. Some of them are Python programmers and are a bit vague about data types, because it all happens by magic in Python. An example of this is all those singletons. They need to access the features easily from the first week. I want them to take an OO approach but they have no idea how to inject dependencies. So I use singletons to begin with and talk about coupling near the end.
2. I’m in a hurry. I don’t get much time to program in my job. For this course, I have 48 hours of preparation time. That seems like a lot of time to write this package. However, during that time I have to write lecture slides, write seminar instructions, respond to student questions, write assessments and write paperwork to explain why I am wasting my time on teaching and programming when I should be writing more paperwork. The time I get left for developing is – well, none at all really. I wrote it over a weekend as a displacement activity from marking student work. A good example is my menu system. It’s hacky because it’s a placeholder I have not had time to fix yet.
3. I’m not that good. I have been teaching C++ for over 20 years, but I get surprisingly little time to develop my own skills. Coding requires a lot of practice. I have picked up a few things over the years, but programming is something I do in my hobby time, not as a profession. If you can see something that needs improving, great, help me out. Send me better code. Just don’t forget that sometimes I do things in an odd way for reason (1) above.

## Compiling the Hornet Engine

### Easy mode

If you are working with Visual Studio 2019 or 2022 on Windows, then you might get lucky. Create an empty folder to put your work in. Download the file “HornetEngineForVisualStudio.zip” and extract it to that folder. Double-click on the file called Hornet.vcxproj. This will open Visual Studio. You should be able to build and run the program. If you are lucky. The menus will work, but the game itself it just a blank screen, because you have not written your amazing game yet. Press escape to get out of it.

Sometimes Visual Studio might want to upgrade the program to a new project or a new version of C++. Go for it. It will be fine, I’m sure.

### Veteran mode

If that didn’t work, or you want to do things yourself, then this read this section. I’m still assuming you are working on Windows with Visual Studio.

Use Visual Studio to create an empty C++ project. If you give it a name like “Thing” you will probably have a folder called “Thing” and another folder inside it, also called “Thing”. Inside that folder will be a file with a .sln extension. Perhaps “thing.sln”. That is where you want to extract the contents of the HornetEngineForVisualStudio.zip file. Don’t copy the .proj file. Everything else, though.

You now need to do several things with Visual Studio:

1. At the top of the screen in Visual Studio, set the configuration to "Debug" and "x64".

**A screenshot of a computer

Description automatically generated with low confidence**

1. Choose Project … Add Existing Item… This will bring up a file dialog, where you can choose all of the .cpp files and .h files and add them to the project. You can do them one at a time, or all at once. **IMPORTANT!!** The file dialog will be looking in the correct folder for your files. If your files are not in that folder, then **DO NOT**, I repeat **DO NOT** use the file dialog to find them in a different folder. Instead use Windows File Explorer to move the files into that folder. I really mean this. Visual Studio will be a pain if the files are not where it wants them. Maybe not today. Not tomorrow. But one day it will ruin your work out of spite.
2. Choose Project … Properties and select the VC++ Directories section. At the top, set the “Configuration” drop down to “All Configurations”. This prevents you from having to do this all over again when you change to a Release configuration. Edit the “Include Directories” line to insert $(ProjectDir)include; at the front. Note there is no space. There is a semi-colon. There is no backslash. The rest of the line may be different to my screenshot below. Don’t worry about it.

For the Library Directories, add $(ProjectDir)lib\x64;

These two lines tell visual studio where to find the header files for SDL and where to find the libraries for SDL. A later section of this guide will explain this.

**A screenshot of a computer

Description automatically generated with medium confidence**

1. A screenshot of a computer

   Description automatically generated with medium confidenceIn the Debugging section, change the “Working Directory” to read: $(ProjectDir)Program

This tells Visual Studio to use the Program folder when the program runs. Files and things that the program uses or creates will be in that folder. If you don’t set this, the default behaviour is to use the same folder as all your source code, which is a pain. It means your file assets and your source code get all confused.

1. In Linker …Subsystem, check that the “SubSystem” is set to “Console”. Actually, it won’t matter much whether you use Windows or Console. SDL2 adapts to either automatically. But I have tested this mainly on Console.

### Hard mode

In this case, you are not working with Visual Studio. Maybe you were conned into believing that Apple products are better because they cost more. Or that Linux is the right thing to use if you have a beard.

You will need to set up your own IDE:

* You will need to download the libraries and header files suitable for your system. You need to download the files for:
  + SDL2
  + SDL-Mixer
  + SDL-Image
  + SDL\_TTF

A quick google search will find these. Unpack the files. You will need to put the header files in a suitable folder and tell your compiler to use them. You will need to put the .lib files in a suitable folder and tell your linker to use them. You will need to put your .dll files in the working directory.

* The working directory for your program will need folders from my zip file: assets, docs and fonts.
* You shouldn’t need to change the code too much, as Hornet does not use Windows-specific code. However, I have used #pragma once in my header files. If your IDE does not like that, you can replace it with the usual #ifndef, #define and #endif structure. I have also use #pragma comment(lib, "SDL2\_mixer") in various places to link to libraries. You may need to delete these lines and handle this through the IDE instead.

## A guide to the folders

Your directory structure will look a bit like this, but don’t worry if you don’t have “Release” folders yet.

A screenshot of a computer

Description automatically generated with medium confidence

Let’s go through them:

**Hornet** – This will be the name of your game. It will contain a solution file (.sln). This is used by Visual Studio to manage various settings. Ignore it. It will contain a project file (.vcxproj). Again this manages various settings and you can ignore it. There may be other files created by visual studio. The important files are the .h files and .cpp files. These are my code files for the engine, and you will add more code files. You can change any of these files – yours or mine.

.**vs** – You may not be able to see this folder. Ignore it. It a Visual Studio thing.

**Debug** – This is used by Visual Studio to compile the code. You can delete it if you want. Visual Studio will just build it again.

**Include** – This is where I put the header files for SDL2 and related packages. The program uses them to compile. You shouldn’t edit these unless you really know what you are doing.

**Lib** – These are the “libraries” for SDL2 and related packages. A library is like a .cpp files that has already been compiled for you. Visual Studio will need these files when it links after compiling. There is a subfolder for x86 and x64, depending on what you are using. It is very unlikely that you will need the x86 folders unless you have a very old computer.

**Program** – This is the “Working Directory”. When the program runs through Visual Studio, it will imagine that the program is located here (even though you won’t actually see the .exe). It contains some .dll files which the program needs to run. A .dll file is a bit like a library – ready-compiled functions, but instead of needing it when the program links, the program needs it when it actually runs. Your program may also create an error log in this folder. If you want to run your program without using Visual Studio, you will need to copy the .exe into this folder and double-click it.

**Assets** – Put your images and sound effects here. There are a couple that my program already uses for menu sounds. Feel free to replace them with better ones and use the same names (or change the code.)

**Docs** – This contains various documents and licences for the packages. An important file is credits.txt. Do not rename this. You can edit the file to add your own credits and they will be displayed by the game program. Yes, I know it is a pain in the arse[[1]](#footnote-1) to have to check the license for assets and make a note of them. If you ever release your game, even for fun, you really should do this. It is annoying to have some great assets that you want to keep but can’t remember where you got them from and don’t want to get sued.

**Fonts** – There are some fonts in here that the engine is already configured to use. You can add your own fonts

**x64** – Will have some subfolders, possible Debug and Release. These will contain various files ending in .obj, .pdb and so on. These are partial compilation files and files used by intellisense. They are often large and can be deleted. Visual Studio can easily rebuild them if it needs them. An important file is the .exe. This is your actual program. If you want to share your program or run it without Visual Studio, you will need to move this into the Program folder. Give the folder to someone and they can run your program.

## Where to program

You should start in the file game.cpp. This is a class that has five functions, which will start mostly empty:

StartOfProgram(). Use this for any code that is needed when the program first runs. This may well be empty.

StartOfGame(). Use this for any code that is needed when the game starts. You will use this to create objects and load assets.

Update(double frametime). This is the main game loop. It will run many times a second. The variable “frametime” is how long (in seconds) since the last time the function ran. You can use this to manage your game physics. Note that in a Windows environment, the game may run much faster than the refresh rate of the monitor. Possibly 2000 frames a second. There are ways to limit this.

EndOfGame(). Use this to delete anything at the end of the game. Typically you use this for delete instructions on anything that used new in StartOfGame().

EndOfProgram( ). Use this to clear up anything that you need to get rid of when the program closes. It may well be empty.

## How to do stuff

### Result

This enum is for defensive programming and debugging. Many of my functions return a “Result”. There are two possible values: Result::SUCCESS and Result::FAILURE. You can use this to check is something worked, and do something about it. You are welcome to use this for your own code if you want, but don’t have to.

### ErrorLogger

The error logger allows you to log error messages if something goes wrong. My own code already does a lot of this. The error messages are helpful and insightful. They may appear in different places, depending on your system. With Visual Studio they may appear in a console window or in the VS Output window or both. They will also be written to a file called error.log in the working directory. The log will stop after 100 lines of messages (because an error every frame can cause 2000 messages every second!) You can change this in the code if you want. Note that the logger will only write messages if you are compiling to “Debug” mode. In “Release” mode nothing will happen.

To write a message:

ErrorLogger::Write("Something has gone wrong.");

There is a version of Write for doubles, but it is usually easier just convert to a string like this:

double huge = 170.0;

ErrorLogger::Write("My IQ is "+std::to\_string(huge));

### Vector2D and Shapes

Vector2D is a geometrical vector. There are functions for finding angles, magnitudes, etc. You can add, subtract, multiply, divide, find a dot product and produce a unit vector. Note that angles are in degrees (0 to 360) and are considered to be clockwise from “north”.

There are five shapes, used mainly for collision detection.

Point2D – This is a simple point in space. Effectively the same thing as a position vector.

Segment2D – A line with a starting point and an end point.

Circle2D – A circle with a centre and a radius.

Rectangle2D – A rectangle aligned to a horizontal/vertical grid.

AngledRectangle – A rectangle tilted to an angle

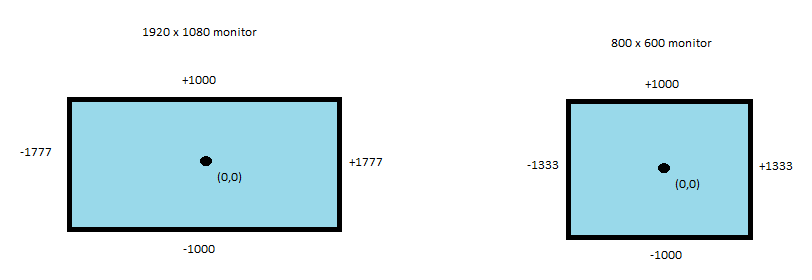
Functions exist to check if two shapes intersect, to find the point of intersection and to find the normal to a surface on intersection. Each shape has various ways to set position and size. See the header file documentation.

### HtGraphics

#### Screen dimensions

The engine will use the native screen resolution when it runs. For example, if your monitor is set to 1920 x 1080, this is what the game will use. However, for most purposes, you will be using the in-game coordinates. These will be the same for all screen resolutions. This means you can create a game that will look the same on all monitors.

The in-game coordinates use the X-value for the horizontal axis and the Y-value for the vertical axis. The centre of the screen is coordinate (0,0).

The top of the screen is set to have a Y value of +1000. The bottom of the screen is set to have a Y value of -1000. The left and right of the screen will depend on the “aspect ratio” of the screen. Wide screens are often around 16:10. Older screens are often 4:3. This means you cannot be sure what the left and right values are. See below for example values on a 1920 x 1080 monitor and a 800 x 600 monitor.

There are functions in HtCamera that can tell you the left and right values of the screen, such as HtCamera::GetRightOfScreen().

Note that using this system, positive values of Y are upwards. This is natural to someone used to using a cartesian graph but may be unexpected to someone used to screen coordinates where the Y-axis it usually inverted.

You can use HtCamera to zoom and pan the camera. This will change the values of the edges, and the centre of the screen may no longer be (0,0). See the HtCamera class for details.

#### Loading images

For most games, you will want to load bitmaps for objects in your game. The engine can load .bmp, .png and .jpg files. It may support other types – try it and see. I recommend avoiding .jpg, since the lossy compression can give undesirable effects with scaling. .png files support transparency which the engine will make use of. By default, any part of the image that is pure black is also considered to be transparent. You can change this with the function HtGraphics::SetTransparentColour().

It is suggested that you put images in the folder called “Assets”.

You can load a file using the function HtGraphics::LoadPicture(). This will return a “PictureIndex”. The PictureIndex is really just an integer, and you will use this later to tell the engine which picture you want to draw. For example:

shipPicture = HtGraphics::instance.LoadPicture(“assets\\ship.bmp”);

or

alienPicture = HtGraphics::instance.LoadPicture(ASSETS+”alien.bmp”);

If you load the same image a second time, the engine will not re-load the image. Instead, it will simply return the Pictur*e*Index of the previously loaded image. This avoids you using a lot of memory for duplicate images by accident. Still, this will still slow the program down a bit. It is best to load all your images at the start of the game if possible, rather than during gameplay.

You do not need to unload images. They will be unloaded when the program exits. In rare circumstances, you may want to manually unload pictures. For example, you may have a program with a large number of large images, and are concerned you will run out of memory. There are two functions for this: HtGraphics::UnloadPicture() and HtGraphics::UnloadAllPictures().

#### Drawing images

You can draw any image by providing the PictureIndex. The function is HtGraphics::DrawAt(). There are various parameters, some of which have default values. The parameters are:

centre – This is a Vector2D that specifies the centre of the image on the screen, using the current settings of the camera.

picture – This is the PictureIndex of the image you want to draw.

scale – This allows to adjust the size of the image. The default value is 1.0. Note that the settings of the camera will also scale the image in addition to this setting.

angle – This allows you to rotate the image. The value is an angle in degrees (not radians) and clockwise is positive. The default value is 0.

transparency – This allows you to make the entire image transparent. A value of 0.5 will be 50% transparent. A value of 1.0 will be invisible. The default value is 0.

Some examples:

HtGraphics::instance.DrawAt( Vector2D(50,50), shipPicture );

This draws the ship picture at coordinates (50,50).

HtGraphics::instance.DrawAt( position, shipPicture, 2.0 );

This draws the ship picture at coordinates in the Vector2D variable “position”. The picture will be double size.

HtGraphics::instance.DrawAt( alienPos, alienPicture, 0.8, 45, 0.5 );

This draws the alien picture at coordinates in the Vector2D variable “alienPos”. The picture is at 80% scale, tilted 45° to the right and 50% transparent.

#### Drawing fundamental shapes

The engine can also draw fundamental shapes – lines, circles, rectangles and pixels. In each case, you will need to specify a colour. The Colour struct allows you to create a colour by specifying ARGB values, with the A value indicating transparency. Each of the values should be from 0 to 255. For example:

purple = Colour(255, 255, 0, 255);

There are various standard colours already create in the engine, for example:

HtGraphics::LIGHTRED;

To draw a rectangle, create a Rectangle2D object in the correct position and run the function **FillRect().** You will need to specify the colour. For example:

Rectangle2D r( Vector2D(400, 500), Vector2D(700, 600);

HtGraphics::instance.FillRect( r, HtGraphics::DARKBLUE );

There are two versions of the function HtGraphics::FillCircle(). One uses a Circle2D object. The other allows you to specify the centre of the circle and the radius instead. In many cases, the game performance may be better if you load a bitmap using LoadPicture() and draw it using DrawAt(). However, FillCircle() does allow you to set the colour programmatically.

To draw a line, you need to specify the start and end. There are two versions of HtGraphics::**DrawSegment().** On allows you to provide a Segment2D and the other allows to provide a Vector2D for the start and a Vector2D for the end.

There are two functions for drawing individual pixels. HtGraphics::DrawPoint( ) and HtGraphics::DrawPointList( ). The first draws a single pixel. The second draws pixels using an array of coordinates, all the same colour. These can be used for some special effects.

#### Transparency

There are three ways to handle transparency in the Hornet Engine. The easiest way is to load an image from a PNG file that has transparency. (Presumably this will also handle semi-transparent areas. No idea. Never tested it. Let me know.)

The second method is to use “colour keying” for transparency. By default, the engine treats any pixel that is pure black as “transparent” when the image loads. This means that if you have an image with a black area around it, you won’t see a black box around the image on screen. A downside is that any part of the image that is *supposed* to be black will be treated as transparent, too. The easy solution is to make those parts nearly black. You can change the colour key to something else (like green) using SetTransparentColour(). You must do this before loading the image, and it is usually best to turn the colour key back to BLACK after loading, or it will make text look weird.

Finally, you can make a whole image transparent at runtime. This is great for making things fade in or out. The final parameter of the DrawAt( ) function is transparency. 0 is not transparent, 1.0 is completely invisible. Use numbers in between for gradual degrees of transparency.

#### Writing text

To write text to the screen, you will need to specify a font. There are five font already loaded. See the credits file for the appropriate licenses. You can load more fonts using the HtGraphics::LoadFont( ) function, but be sensitive to intellectual property rights. When you use the LoadFont( ) function, you will need to specify the “point size” of the font. If you want different sizes, you can load the font multiple times. It is usually easiest the just load a large point size and scale the text when you draw it. When you load a font the function will return a “FontIndex” which you can use to specify the font you want. For the five fonts already loaded, you can just use the numbers 0,1,2,3,4.

The easiest way to draw text is to use the functions **HtGraphics::WriteTextCentred()** and **HtGraphics::WriteTextAligned().** The difference between the two is that “Centered” means that you will provide the coordinates for the centre of the text on the screen. This might be useful for situations where you want to add a label to an in-game object. “Aligned” means that you provide the coordinates of the top left corner of the text. This is more useful for GUI elements, such as a HUD. There are actually two versions of WriteTextAligned() – one uses a Vector2D, the other uses 2D coordinates.

There are also variants for writing integers and floating point numbers: **HtGraphics::WriteIntCentred()** and **HtGraphics::WriteFloatCentred().** Plus “Aligned” versions.

For all of these, there are parameters for the coordinates, the text (or the number) and the font to use. You should provide a FontIndex, or a simple integer. There are optional parameters for the scale and the angle (the angle is only for “Centered” versions.)

Some examples:

HtGraphics::WriteTextCentred( position, “Danger”, HtGraphics::RED, hazardFont, 0, 0.8);

HtGraphics::WriteIntAligned( 800, 950, score, HtGraphics::LIGHTBLUE, 2, 0.5 );

(In this last example, I am using “2” for the font. This will use the third pre-loaded font.)

These functions are (relatively) slow. Each time, the engine needs to create a new textures, draw it and delete it. If the text or the number changes frequently, this is unavoidable. However, for text that will not change during the game, there is an alternative method.

The second method is to create a picture using the text, font and colour at the start of the game. This can then be drawn as a normal picture using the PictureIndex. The function for this is HtGraphics::CreatePictureFromText(). Provide the text, the font and the colour. The function will return a PictureIndex. You can draw this as normal using HtGraphics::DrawAt(). There are no functions to create a picture from integers or floats, but it is easy to convert to strings using standard C++ functions.

CAUTION CAUTION CAUTION. There is a “gotcha” here. If you create a picture from the same text again, this will create a new picture. The engine will not realise that a suitable picture already exists. It is possible to make a mistake and create thousands of copies of the same picture, wasting system resources. To avoid this, only create a picture from text at the start of the game, not during gameplay.

A third method for handling text is to create a bitmap using an image editing tool. This allows you to create funky and exciting text. With a bit of work, you can create your own text engine, using a bitmap with all the letters of the alphabet. Ask your instructor for guidance.

#### Presenting

When you draw a scene, you are actually drawing to a "back buffer", not the screen itself. At the end of each frame, you need to "present" this back buffer, so that it appears on the screen. At the same time, you get a new back buffer to draw on. (In fact the new back buffer might actually be what used to be the screen – the system just swaps between the two. Although for complicated reasons, there may be more than two – double- or triple-buffering.) Presenting the back buffer is sometimes called "flipping", perhaps because it is like flipping the pages of a flip picture book. At the end of Game::Update(), you should have a call to HtGraphics::Present(). You may sometimes want to call this elsewhere, for things like menus, but mostly you can just leave that line of code at the end of Update( ) and ignore it.

#### Setting background colour or texture

By default, the background colour is black. When a new back buffer is created, it is filled with this colour. You can change the background colour using HtGraphics::SetBackgroundColour( ). Note that the background colour is not the same thing as the colour key for transparency, even though they are both black by default. To change the transparency colour, see the section on transparency.

It is also possible to use a texture or image as a background. Load the picture as normal using HtGraphics::LoadPicture( ) and send the PictureIndex to HtGraphics::SetBackgroundTexture( ). The background image will be stretched to fill the screen. If you want something else, like tiling, you can either ask your instructor or modify the HtGraphics::Present() function.

#### Getting information

If you need to know the resolution of the screen, you can use the functions HtGraphics::GetWindowHeight() and HtGraphics::GetWindowWidth(). These give you the size of the window in pixels. What you are more likely to need is the width and height in in-game units. There are functions for this in the HtCamera class.

#### Releasing images

Images are released when the program exits. You don't normally need to release images at other times, but if you are doing something complicated:

To release an individual picture: use HtGraphics::ReleasePicture().

To release all picture: use HtGraphics::ReleaseAllPictures().

It is also possible to release all fonts, but you are not likely to need that.

### HtCamera

The HtCamera class allows you to pan the view around a game world, and also zoom in and out. At the start of the game, the camera is placed to that the coordinates (0,0) are at the centre, and the screen has a vertical height of 2000 game units. (From -1000 to +1000).

Note: Normally, the camera area and the screen area are the same thing. If an object is on-camera, it is also on-screen. However, there is a function to switch the camera "off". You can still "move" the camera around, but this will have no effect on the screen. Also, objects that are on-camera might not be visible on the screen, in this situation.

#### Moving the camera

You can move the camera to another location using the function HtCamera::PlaceAt( ). The function accepts a Vector2D. The new camera position will set the centre of the screen to the position described by that vector. For example:

HtCamera::instance.PlaceAt( Vector2D( 200, 500) );

Will place the camera so that the centre of the screen is now (200, 500)

HtCamera::instance.PlaceAt( jellyFishPosition );

Will centre the camera on the jellyfish.

HtCamera::Reset( ) will return the camera to the default position (and zoom).

#### Zooming

To zoom in and out, use HtGraphics::SetZoom(). The default zoom is 1.0. If you use larger numbers, objects on the screen will get larger. I don't know what will happen if you use negative numbers. Probably nothing good.

HtCamera::Reset( ) will return the camera to the default zoom (and position). Or you can just use SetZoom(1.0).

#### Getting information

It is often useful to know the limits of the screen in in-game units. There are four functions to tell you the limits of the camera area at top, bottom, left and right:

HtCamera::GetTopOfCameraArea()

HtCamera::GetBottomOfCameraArea()

HtCamera::GetLeftOfCameraArea()

HtCamera::GetRightOfCameraArea()

The function HtCamera::GetCameraArea( ) returns the same information as a single Rectangle2D. Testing for collision with this will tell you whether and object is on-camera.

You can check if a position vector is on-camera, using HtCamera::IsOnCamera( )

Note that all of these functions are working on the camera's area. If you have set UseCamera(false), this will not be the same as the screen area.

There are functions to request the current camera position and zoom: HtCamera::GetZoom() and HtCamera::GetCameraCentre().

There are functions that can transform coordinates and shapes from native screen coordinates to world coordinates and vice versa. You will probably not need these, but the function NativeTransform() converts from game-world shapes to native screen shapes. The function GameTramsform( ) does the opposite.

#### HUD elements and the camera

When drawing HUD data such as the score and pointers, you often don't want these objects to move around when the camera does. You can temporarily disable the camera using HtCamera::UseCamera(false) and re-enable it after drawing with HtCamera::UseCamera(true) . This is different to using Reset( ), as the settings are not lost, and camera functions still assume the camera is in it's previous position.

### HtAudio

HtAudio can play music and sound effects. Music is handled slightly differently to sound effects. You can load, play and stop music, but there is only ever one music file currently loaded. If you want to return to an earlier music track, you will need to load it again.

Sound effects work by loading a list of sound effects, which remain loaded in memory until they are cleared – either all at once or individually. Each sound effect is identified by a "SoundIndex", which is really just a number.

It is perfectly possible to load a music file as a sound effect, or even a short sound effect as a music file. They are just treated differently by the class because they are used in different ways.

The system can play up to eight sound effects simultaneously. (This may differ on some platforms, and may change with improvements to SDL.) Only one music file can play at a time.

HtAudio is a singleton and functions can be accessed using HtAudio::instance. For example:

int channel = HtAudio::instance. Play(explosionSI);

#### Loading and playing sounds

You can load a sound effect using SoundIndex LoadSound(std::string filename); The file can be a .wav or .mp3 file and will normally be placed in the assets folder. The function returns a SoundIndex, which should be stored in a variable as it will be needed later when you want to play the sound. For example:

SoundIndex explosionSI = HtAudio::instance.LoadSound(ASSETS+"bang.wav");

Here I am using a constant which stores the folder name of the assets folder, but you can just do it manually if you want: LoadSound("assets\bang.wav");

If you load the same file a second time, HtAudio will simply return the SoundIndex of the sound effect that was previously loaded. This means you are safe to "load" the same sound effect as many times as you want without wasting memory.

To play the sound, use the Play( ) function. You will need to provide the SoundIndex of the sound effect:

int channel = HtAudio::instance.Play(explosionSI);

The integer "channel" may not be needed. When a sound affect plays, it is given another number to indicate what "channel" it is playing on. The same sound might be playing several times, on different channels. (For example an explosion sound might be playing three times during an intense bit of gameplay on channels 4, 6 and 7.) The "channel" allows you to stop a sound effect early, or to adjust the volume while it is playing. If you don't need to do that, there is no need to store the channel.

#### Looping sounds

Normally, sounds play once, then stop. You may want to ask a sound effect to play repeatedly. In this case, it is strongly recommended that you store the channel number. If not, you won't be able stop the sound. Ever. It will follow you around for the rest of your life. To make a sound loop, add the word "true" to the function call:

int engineChannel = HtAudio::instance.Play(engineSI, true);

(In fact, the sound will actually stop after playing several thousand times, but this is usually not relevant. In fact, SDL has the ability to make a sound repeat a set number of times. I have not implemented that in the engine, but it would be trivial to include.)

#### Stopping sounds

If you want to stop a sound effect while it is playing, you need to use the channel number. This is particularly useful for looping sounds, but any sound can be stopped.

HtAudio::instance.Stop(engineChannel);

#### Loading and playing music

Music is a bit easier, as you don't have to deal with a SoundIndex or channel. You can load music with LoadMusic(). Play it with PlayMusic( ) and stop it with StopMusic( ). You can also pause music and resume from the same point using PauseMusic( ) and ResumeMusic( ).

#### Adjusting volume

There are three ways to set a sound volume: master volume, by sound or by channel. They combine, so if they are all set to 0.5, the sound will play at 0.5 x 0.5 x 0.5 = 0.125. The system volume setting will also affect this. The volume can be set from 0 (silent) to 1.0 (full volume). If you go outside these values, it won’t cause any problems- the functions will just raise the number to 0 or drop it to 1.0.

Master volume : Use SetMasterVolume(double volume)

This will set the volume for all sounds but WILL NOT affect sounds already playing and WILL NOT affect music.

Sound volume: Use SetSoundVolume(SoundIndex sound, double volume) and provide the sound index of the sound. This will set the future volume of the sound whenever it is played. It WILL NOT affect sounds currently playing.

Channel volume: SetChannelVolume(int channel, double volume) Provide the channel number. This is the tool to use for sounds that are already playing. It is particularly useful for looping sounds.

Music volume is completely independent of this. Just use SetMusicVolume(double volume). This will affect both music that is already playing and music that is yet to be started.

#### Panning

Making the sound pan left and right is similar to setting volume. It can only be done for a channel. Set a number from -1.0 (full in left speaker) to 1.0 (full in right speaker). The functions clamp the number if you provide a number larger than 1.0 or smaller than -1.0.

#### Releasing sounds

Normally, you can leave sounds in memory, and they will be released when HtAudio shuts down at the end of the program. If you have a lot of large sounds, you may need to release them when they are no longer needed, to free up memory. If you need this, you can release ALL sounds using ReleaseAllSounds( ) or release a specific sound using ReleaseSound( ).

### HtKeyboard

HtKeyboard allows the game to get input from a keyboard. It might or might not make an on-screen keyboard appear if you don’t have a physical keyboard. I have not tested that.

The system can poll the current state of keys, support a keyboard listener and can track typed messages.

HtKeyboard is a singleton and functions can be accessed using HtKeyboard::instance. For example:

bool hit = HtKeyboard::instance.KeyPressed(SDL\_SCANCODE\_SPACE);

#### Polling

You can check if a specific key is currently up or down using KeyPressed. You will need to provide a “scancode” for the key. Examples are:

SDL\_SCANCODE\_0

SDL\_SCANCODE\_F1

SDL\_SCANCODE\_A

SDL\_SCANCODE\_ESCAPE

SDL\_SCANCODE\_LSHIFT

See the header file for more examples.

If you want to know if a key has been pressed for the first time on this frame, use NewKeyPressed() .

#### Typed messages

You can allow the user to type a message using several keys. This could be useful for entering data into text fields, using console commands, key combos or cheat codes. Note that the system will not display the keys strokes – you will need to do that using HtGraphics.

To start text input (for example when the user clicks on a text box), use:

HtKeyboard::instance.StartTextInput();

To find out what the user has typed so far (for example, to display it on the screen) use:

HtKeyboard::instance.PeekTextInput();

To stop text input use:

HtKeyboard::instance.StopTextInput();

This will also return the text typed.

The system does not support complicated text composition such as backspaces or pasting.

#### Keyboard listener

The above systems use polling. I.e. you will be checking the key pressed every frame. Alternatively, you can create a HtKeyboardListener to receive key events. Any class can be made a keyboard listener by inheriting from HtKeyboardListener. For example:

class Spaceship : public IHtKeyboardListener

{

That class will need to implement the functions:

void HandleKeyboardEvent(const SDL\_Scancode& SDLkey,

KeyboardEventType keyboardEventType)

These will be called when a key is pressed (or released).

Once creating the class, you will also need to register an object of that class with HtKeyboard. This could be done in the object’s constructor for example:

Spaceship()

{

HtKeyboard::instance.RegisterKeyboardListener(this);

}

At the moment, HtKeyboard will only track a single MouseListener object. This would be fairly easy to change if you want to fiddle with the code.

### HtMouse

HtMouse gives information about mouse position, movement and mouse buttons. It supports three axes: left/right, up/down and mouse wheel forwards/backwards. It supports up to three buttons: Left, right, and middle.

HtMouse is a singleton and functions can be accessed using HtMouse::instance. For example:

Vector2D mousePos = HtMouse::instance.GetPointerWorldPosition();

#### Mouse position and movement

HtMouse uses the system mouse pointer, but it actually has three positions, depending on whether you are thinking about the native mouse position, the screen position or the world position.

* Native position. This is the position of the mouse pointer as far as the operating system is concerned. Normally (0,0) is the top left and the Y-axis goes DOWNWARDS. The maximum values of X and Y depend on the screen resolution. You probably won’t use this much, as the rest of the Hornet system does not use these native coordinates. You can get the values by using GetPointerPositionX() and GetPointerPositionY() . Also GetPointerPositionZ() for the mouse wheel.
* Screen position. This is the position of the mouse pointer using the screen coordinates in Hornet. (0,0) is the middle of the screen, and the Y-axis goes upwards for positive movements. The top of the screen is Y=1000. The bottom of the screen is Y=-1000. The X limits depend on the screen aspect ratio, but are usually about +-1775 for a wide-screen monitor. You can get this position using GetPointerScreenPosition(), which returns a Vector2D. This is mainly useful when you want to click on HUD elements.
* World position. This is the position of the mouse pointer using the world coordinates in Hornet. Using the HtCamera, you can move the viewport around a game world. You can get find the position of the mouse in the world using GetPointerWorldPosition(), which returns a Vector2D. This is mainly useful when you want to click on in-game elements. If you have not moved the camera, the world position will be identical to the screen position.

It is possible to force the mouse pointer to any position, using SetPointerPosition, SetPointerScreenPosition or SetPointerWorldPosition.

#### Mouse clicks

You can determine if any of the mouse buttons are up or down using IsMouseDown . You will need to specify which mouse button, using HtMouseButton::LEFT, HtMouseButton::RIGHT or HtMouseButton::MIDDLE. The function will return true if the button is down, or false if it is up.

Often, you want to know if a mouse button has been clicked for the first time. Use the function IsNewMouseDown for this.

#### Mouse movement

If you want to know how much the mouse has moved this frame, rather than where it is, use GetMouseMoveX(), GetMouseMoveY() and GetMouseMoveZ(). These return the movement in native units, so moving the mouse forwards is an upwards movement, giving a negative movement. This is useful when you want to use the mouse to control something, without a visible cursor (like a virtual joystick), as the mouse movement is not limited by the edges of the screen.

There is an issue here. If you move the pointer to the edge of the screen and try to move the pointer beyond it, you won’t get a mouseMove in that direction. The move will be zero. However, if the mouse pointer is invisible (see below), you will get a movement in this case. This is useful if, for example, you are using the mouse as a joystick or a camera control. You can move the mouse as far as you want and still get movement in the code, without being limited by the size of the screen.

#### Mouse appearance

The mouse pointer will use the system pointer. You can hide or show the pointer using SetPointerVisiblity(). If you want a custom pointer, there are two options. One is to persuade the operating system to use a different pointer image. This will be system-dependent. The other way is to simply hide the mouse pointer and draw an image in the right location using HtGraphics::instance.DrawAt().

#### Mouse listener

The above systems use polling. I.e. you will be checking the mouse position every frame. Alternatively, you can create a HtMouseListener to receive mouse events. Any class can be made a mouse listener by inheriting from HtMouseListener. For example:

class Spaceship : public IHtMouseListener

{

That class will need to implement the two functions:

void HandleMouseMove(int x, int y, int z);

void HandleMouseButtonEvent(HtMouseButton button, bool down);

These will be called when the mouse moves or when a button is clicked (or released).

Once creating the class, you will also need to register an object of that class with HtMouse. This could be done in the object’s constructor for example:

Spaceship()

{

HtMouse::instance.RegisterMouseListener(this);

}

At the moment, HtMouse will only track a single MouseListener object. This would be fairly easy to change if you want to fiddle with the code.

The HandleMouseMove function received native x,y,z movement values, which may not be useful. However, you can use the function to be informed that a movement has happened and just use HtMouse::instance.GetPointerScreenPosition() to find out where the pointer actually is.

### HtGameController

HtGameController is a class that receives information from a game controller device, such as a joystick or joypad. It can handle any number of axes or buttons.

HtGameController is a singleton and functions can be accessed using HtGameController::instance. For example:

double up = HtGameController::instance.GetAxis(1);

#### Polling axes

Controllers have at least two axes, but usually more. Each axis has a number 0,1,2,3 etc. 0 is normally the X axis and 1 is the Y axis. To get the current position of any axis, use GetAxis( ) and specify the axis number. This returns a number from -127 to +128, with 0 as the centre. Note than many devices have a “dead zone” at the centre, so being close to the centre will still give 0.

For convenience, there are also functions GetXAxis(), GetYAxis() and GetZAxis()

#### Polling buttons

Controllers have many buttons, numbered 0,1,2,3 etc. You can get the current state of any button using IsButtonDown( ) **.** Provide the button number. It will return true if the button is currently down. If you want to know if a button has been pressed for the first time this frame, use IsNewButtonDown() in the same way. Note that “buttons” like shoulder buttons with a range of movement are actually treated as axes. In some controllers, they might be both on/off buttons and axes.

There is no listener system for joypad events. For a game controller, simple polling is usually easier.

#### Multiple controllers

The user may have more than one controller (or none). You can set the current controller using SetCurrentController(int number) . The “number” will be 0,1,2, etc. If there is only one controller, the number should be zero. This is the default controller.

#### Getting information

There are various functions to get information about controllers:

GetNumAxes() - Returns the number of axes on the current controller.

GetControllerName() - Returns the name of the current controller

GetNumButtons() – Returns the number of buttons on the current controller

IsControllerAvailable() – Returns true if a controller is active. If this is false, you may have no controller connected or have tried to set a non-existent controller.

GetNumControllers() – Returns the number of controllers currently connected.

### HtUnifiedInput

This is a system that combines inputs from Keyboard, Mouse and GameController, and allows the player to reconfigure controls easily from the menu. When I get round to writing it, it will be amazing.

### HornetMenus

You don’t need to make any changes to HornetMenus. However, you may want to customise the appearance or change the sound effects. The sound effects are loaded at the bottom of Initialise( ). To change the appearance of the menus, look at UpdateMain() and UpdatePause(). Remember that the Credits menu takes the data directly from the credits.txt file.

If your program takes a while to load assets, you may notice a “Loading” screen. If you want to customize it, look at DisplayLoadScreen(). Making an animated load screen will be more difficult. You will either need to draw at intervals while you are loading the files, or you would have to start messing with threads. Either way, you will not be using the DisplayLoadScreen( ) function and will make your own.

The whole menu system is currently a bit hacky. I’ll refactor it when I get time.

### HornetApp

This manages the main loop. There is no need to mess with this code unless you want to do something really clever. There are some constants at the top of HornetApp which you might want to edit if you use a different folder structure. The constant APPNAME is the name of your program. It appears on the main menu and on the window title (which is normally invisible anyway.)

### Settings

The Settings singleton class provides a basic way to save and load settings. It will be automatically loaded at the start of the program and saved at the end. At any time, you can use Settings::instance to read and change settings. You can also add new settings. Each setting requires a name and a value. Each value should be either a string or a floating-point number. (You can simply use casts to use integers or Boolean values.) The settings will be stored in the working directory, with filename settings.dat. This can be manually edited. To read the value of one of the settings use the member function GetStringValue(string name) or GetNumberValue(string name) . For example, if you have a setting called “maxspeed”, you can load this using

int speed = Settings::instance.GetNumberValue(“maxspeed”);

The name of each setting is case sensitive and should not contain whitespace. In Debug configuration, the system creates an entry in the error log if you request a setting that does not exist.

To change a setting use SetValue(string name, string value) or the overload SetValue(string name, double value).

You can also use the same member functions to create a new setting, with a new name. It will be saved at the end of the game, along with the other settings.

There are functions to load, save and clear the settings, but these are used already by the engine, and you should not need to call them manually.

It will be inefficient to read the value of settings on each frame. It is best to read the value when the game starts and store this in a suitable variable.

### Angle

This is a class to manage an angle. It is just a wrapper for a double, but restricts the value to the range [0,360). It supports basic mathematical functions and free casting to and from doubles. The main value is to return the shortest angle difference between two angles using the function Difference( Angle other ).

### GameObject

The GameObject is an abstract superclass for concrete game objects. The class supports:

* File loading. The member function LoadImage can be called to load one or more files. It can be called multiple times to load several files, supporting animation.
* Rendering. The default Render function will draw the file image, using protected variables m\_position, m\_scale, m\_angle, m\_transparency, m\_imageNumber.
* Update. The abstract Update( ) function should be overloaded in the concrete subclass.
* Collision. Override GetCollisionShape() to specify the collision area of the object. Override ProcessCollision() to specify collision behaviour. For collision to be detected, m\_collidable should be set to true in the constructor, before calling Lock().
* Type. Each GameObject has a constant TYPE which is specified in the objecttypes.h file. If a type is not specified at object creation, the type will be “UNKNOWN”.
* EventHandling. Objects can receive Event objects with the HandleEvent( ) function. For events to be received, m\_handleEvents should be set to true in the constructor, before calling Lock().
* Multiple scenes. When an object is created, it can be assigned to a “scene number” before being added to the ObjectManager. The default is 0. Objects with a scene number that does not match the current scene will not update, render, or collide. Objects will still receive event messages if they are not in the current scene.
* Automatic destruction. On object will be destroyed by the ObjectManager at the end of the frame if m\_active is false.
* Debugging. If the program is compiled to a Debug configuration, objects can report Debug information in the RenderDebug function. You can override this, but should call GameObject::RenderDebug( ) at the start of your own function. Report debug information using AddDebugLine ( ).

### ObjectManager

The ObjectManager manages a list of GameObjects. Each frame, all objects in the current scene will update, render and check for collisions. At the end of the frame, all objects that are not active (m\_active == false) will be deleted. Events sent to the ObjectManager (using HandleEvent) will be passed to all GameObjects that have registered for events.

If the program is compiled to a Debug configuration, the system will allow the user to view runtime debugging information for a selected object. This can be activayted and deactivated with the HOME and END keys. The currently selected object can be cycled with PAGEUP and PAGEDOWN. The program can be made to run very slowly using INSERT.

### Event

Event is used to send and receive event messages. For each event, specify:

* pSource: A pointer to the GameObject sending the event.
* type: The type of event. Possible values are specified in event.h
* Vector2D: The location of the event, if relevant.
* Data1 and Data2 – Allow for additional data if needed.

## Memory Leaks Detected

If you run the program in DEBUG configuration using Visual Studio, you may get told that there are memory leaks. This means you messed up. Messages could appear in various places:

A black background with white text

Description automatically generatedIn the console after the program completes:

In the error.log file:

A screenshot of a computer error

Description automatically generated

In Visual Studio’s output window:

A screenshot of a computer screen

Description automatically generated

You will usually get an alert in all three places.

This usually means that somewhere you have used “new” without using “delete”. Check standard texts on C++ for information about memory leaks, or consult your tutor who should have taught you better.

It can be difficult to find the cause of memory leaks, which is why it is best to deal with them as soon as they appear. If you just wrote five lines of code and the message started to appear then the problem is (probably) in those five lines of code. If you have just written two hundred lines of code, then you have a problem.

The dump that appears in Visual Studio’s output window may help a bit. Mainly it tells you the size of each memory leak. In the image above, the first memory leak is four bytes long, so it may be an integer. The second one is 16 bytes long. Perhaps an object that contains two doubles? Such as a Vector2D? Or maybe an object that contains four integers? Such as a Colour? Programmers love a mystery, especially at 2am with a deadline.

However, there is an issue. SDL2 creates some memory leaks. This is unavoidable, as SDL2 loads other libraries, such as DirectX and OpenGL. These packages often leave small memory leaks on initialisation and shutdown. These leaks will (probably) not trigger the “Detected Memory Leaks” message but may appear in the dump. The leaks that are listed first in the dump are (probably) your fault. Later ones are (probably) not.

1. Yes, I’m English. I think an ass is a donkey. [↑](#footnote-ref-1)