Portfolio Project 02

Section 2 contains two Portfolio Projects. You are currently undertaking Section 2 Part C, which contains Portfolio Project 02.

You should NOT submit your work for marking until you have completed ALL Portfolio Projects contained within Section 2. Therefore you must submit your work to your tutor team for marking once you have completed all of the projects.

As you complete the work for each of these projects you should place your work in a suitable folder.

Details regarding how to submit your portfolio work for marking can be found at the end of this document.

This is the second of a series of special projects that will test your ability and understanding in specific areas of game programming. These will be assessed by your tutors and must be completed to a sufficiently high standard to allow you to progress to the next section of the course.

You must pay particular attention to the presentation of your project; this will be an area that we will be looking at during the assessment of your work.

It is recommended that you use the previously supplied Visual C++ Express Edition to create this portfolio project.

Your submission must include:

- A .zip or .rar file containing your source code.
- A compiled executable for a Windows environment.
- A document in .doc or .docx format listing any idiosyncrasies or bugs in the code and crediting the source of any non-original code used.

Learning Objectives

- **Demonstrate the ability to prepare and execute your own code.**
- **Demonstrate skill in investigating programming tasks within a game framework.**
- **Demonstrate ability to develop key games software systems that can be reused as part of a larger game or demo.**
- **Be able to Understand common techniques used in writing games software.**

The Brief

For this project you will be working on a series of assignments and solutions designed to be integrated into your own game design project that you have written. These tasks should enable you to build a frame work with which to integrate into your own game. You are encouraged to source and use your own art and sound assets, many of which can be found for free on the Internet. Remember to always check that the assets you use are free to use and are not subject to copyright law.

Scope

This project is split into five separate programming tasks each of which should take you approximately 48 hours to complete. Each task will consist of a programming exercise and you are expected to provide for each task, a working executable, source code and a document of no more than 2000 words. The document should detail how you approached the problem any key features of the code that you think need special consideration and crediting any third party source code or libraries that you have used.

You need to make sure that each solution solves the programming task and that you have documented how you approached the problem.

It is expected that to achieve full marks you will provide evidence of having researched the particular problem and have implemented your own software solution. Your software should also be robust and free from any obvious serious bugs.

Tasks

Path Finding

Many games require realistic behaviour from objects that are not under the direct control of the player. This could be the dynamic movement of physical objects under the influence of external forces such as gravity, but it could also be the *artificial intelligence* of characters and their apparent decision making abilities during navigation. This *path finding* behaviour allows an object to travel between two predetermined points while avoiding obstacles along the way.

A game level is typically represented by a map, where each indivisible element equates to a navigable point that can be occupied by a game object. These *nodes* do not necessarily need to be equivalent to the on screen graphics, but it can be simpler to visualise the problem in this way. The *graph* of nodes represents all possible connections between each point and will (somewhere) contain the path or paths with the lowest *cost*. Finding this shortest route efficiently is the purpose of all path finding algorithms.

A popular path finding method is the *A** search algorithm, although many similar routines exist. This project aims to investigate path finding within the context of a simple game framework. The game is not important, but it must at least present the need for objects to navigate around a level that offers challenges such as goals and obstacles. The game could require a number of objects to navigate a simple maze, but a variable terrain map would also offer significant exploration opportunities.

The basic idea is to find the path with the lowest *cost*, which in a simple maze may well be the shortest route too. However, a node could also be assigned its own cost value, making it more or less desirable to the path finding algorithm. The A* algorithm employs a *heuristic* function to estimate the remaining cost of the route to the goal, which is a good guess, but probably doesn't take obstacles into account. Path finding is computationally expensive, so it will be necessary to investigate various optimisations when many objects are navigating at once.

Depending on the distance between the start and end points, it is possible to decrease the search time by using a hierarchical system that finds a coarse route based on higher level nodes, only to solve the fine detail over the short path found within the larger nodes. Similarly, predetermined *waypoints* can be added at the level design stage that are used to precompute connectivity information. *Collision avoidance* between objects tries to prevent paths crossing so that objects appear to be aware of each others' presence and take action ahead of time, which is particularly useful in crowd simulation.

Sprite Blitting

Traditional games using 2D graphics have to rapidly display numerous *bitmaps* on screen to give the illusion of movement and animation. A full screen background layer may be represented by *tiles*, which are stored as an indexed array of small repeatable bitmaps. The objects displayed in front are *sprites* and are also stored as small bitmaps. Some platforms provide direct *hardware* support for these components, freeing the game code from the need to construct the *raster* image. However, it is important to understand (and sometimes necessary to implement in *software*) how the graphical display of a game can be constructed from these smaller components.

A software implementation of sprites typically requires a *framebuffer* that is a bitmap representing the entire screen. It is a two dimensional array in memory, given a width and height in *pixels*, with each element defined by a *pixel format*. This format is usually governed by hardware constraints, but typically contains at least the three red, green and blue *colour channels* to some *bit depth*. For example, a 16 bit framebuffer may use a pixel format that gives 5 bits (32 levels) per channel, or an 8 bit *indexed colour* buffer may reference a 256 entry *palette* of 24 bit colours. The sprites themselves refer to bitmaps, which may have a different pixel format and are to be copied to the framebuffer at a certain *coordinate*.

This operation is known as a *blit*, which basically copies the contents of one memory location to another, with regard to the relative pixel format, *stride* and *compression*. This assignment aims to develop a complete software sprite implementation within the framework of a simple game. Emphasis must be placed on the techniques used to render the sprites to the screen, rather than the game itself. Existing APIs can be used for loading image data and for the display and update of a window. The sprite blitting should be platform agnostic as far as possible and capable of handling at least truecolour and indexed colour pixel formats.

A common technique for smooth animation is to use a *double buffer*, where two framebuffers exist to prevent *tearing* artifacts. While the front buffer is being displayed to the player, the game is blitting into the back buffer ready for the next frame. During the *vertical black* interval, the back buffer then becomes displayed to the player and the process continues to the next frame. A hardware implementation may require both *pages* in *video RAM* and then *flip* between them each frame. In software, it is possible to have the back buffer in main memory which is then itself blitted to the front buffer during the vertical blank.

Transparency may also be a requirement of some pixel formats, requiring the blitting function to either ignore or blend certain pixels based on their colour value, using techniques such as *colour keying* or *alpha blending*. Animation can be achieved by moving the foreground sprites over the background tiles, but it is usually expected that each sprite also displays different *frames* of animation over time. These can sometimes be stored efficiently in the same bitmap, with coordinates used to locate the offset of each frame. Image compression techniques, such as *palettisation* and *run length encoding* can be employed to not only decrease the memory requirements but also increase the blitting speed in some cases.

Data Serialisation

Game assets, such as bitmaps and samples for sprites and sounds, can usually be found alongside the executable binary game file on the disk. This data is dynamically loaded at run-time so that it can be accessed by the game code. It is stored as a stream of bytes that must be *parsed* in order to construct the internal data structures of the game. The process of converting the structure and data of an object to and from a stream is known as *serialisation*.

Common file formats for bitmaps and samples usually have their own well defined file formats and serialisation libraries. This may well be useful for the initial development a game, however there are many reasons why it may be beneficial to use a custom format specific to the game, such as *compression* (increasing speed while decreasing size) or *obfuscation*. Also, there will be data unique to the game for which no file format exists, such as levels, maps, animation or configuration files. For this reason, it is necessary to develop a data serialisation framework that is capable of loading and saving game objects.

Another important aspect of game development is content creation and the ease with which others can supply usable data for the game. While widely available tools can produce data in common file formats, custom data must either be created by in-house tools or by hand in a *human readable* text format. The aim of this assignment is to produce a simple game example that makes extensive use of its own data serialisation mechanisms. A complete game is not required, but it must demonstrate the ability to load and save custom game data using a file format that can easily be edited as text. The ultimate goal is to implement a *serialiser* class and an abstract *serialisable* class that can be used to allow an object to supply named values to the serialiser.

A class that implements the serialisable class will provide the load and save methods which are invoked from the serialiser. The load method will be able to call methods on the serialiser to register each of its members with the serialiser, providing a type, name and reference or a function pointer to a handler for complex types. As the serialiser parses the file, it will gradually initialise members of the object. The save method will be able to call similar methods on the serialiser to write members to the stream.

The serialiser can convert the data to and from a custom human readable text file. However, there are many existing formats such as *XML*, *YAML* or *JSON* which are more than adequate for the task. Ideally, the serialiser itself would also be an abstract class that can then be used to implement serialisers for more than one data format. The ability to serialise a binary file would then be possible, but care must be taken in order to provide platform independence regarding type conventions such as *endianess*, size and *character encoding*.

Procedural Generation

While the majority of game data is created during the development process and eventually serialised from disk, there are occasions when it is necessary to generate the data *procedurally* at run-time. Some data may just be too large to distribute or fit in memory, while other data may depend on user input such as customising in-game parameters. The general case for procedural generation is when a large set of combinations is required from a limited choice of parameters. Game code then applies the parameters to an *algorithm*, possibly using existing assets to produce one of many new unique assets.

There are numerous examples where this technique can be put to good use in a game. The generation of random levels is paramount to many non-linear games from puzzle to strategy games, to provide a unique, unpredictable environment on each replay. There may be many instances of the same object in a game where it would be beneficial provide slight variations in their appearance in order to improve realism. Apart from producing a variety of seemingly unpredictable content, it is also important to realise that the same set of parameters will always produce the same outcome. This is a very useful form of data compression when it comes to serialising game data.

This assignment expects some form of procedural generation of game assets within a simple framework game. The game can be minimal, but it must demonstrate an innovative approach to how it produces some of the content. The level or even the sprites themselves are prime examples of objects that can be generated on the fly. The parameters could be randomly assigned or they could be defined in a configuration file. It is only necessary to present the ability to produce a large variety of game assets from a limited set of building blocks.

Efficiency in terms of size and speed is a key component of a procedural generation algorithm. The game may choose to produce a large static data set (such as a map) at the beginning of the game, in which case it has the time to generate this during a slower initialisation phase. But the game may instead require an extremely large (practically infinite) play area where only a small portion is available to the player at any one time. In this case, the algorithm must be capable of producing the procedural data in a very short amount of time, based on coordinates, to give the impression of a seamless world.

The algorithm itself will likely be based around some common mathematical concepts such as *probability*, *noise* or *fractals*. A simple *pseudorandom number generator* given the same seed parameter will always produce the same sequence of apparently random numbers. This sequence can then be used to compare against probability parameters, to test the chances of a certain game object appearing at a specific location of the map, for example. The same pseudorandom sequence could also be used as the basis of a noise function which may be used to represent a landscape. Fractal algorithms can be used to produce data that exhibits self similarity at many levels that may not otherwise be possible with other techniques. Often, a hybrid approach may offer the best solution, an obvious example of which would be the simulation of a tree.

Memory Management

Game data must at some point occupy a section of main memory so that the game code can access it after it has been loaded from the disk. This can occur at load-time when the game code itself is loaded from disk into main memory and *statically* defined data structures are allocated a region of their own. This memory is deallocated when the game finally exits. It can also happen *automatically* from the *stack*, such as the allocation of parameters, local variables and return value during a function call. This memory is deallocated simply and quickly when the program flow returns from the function call.

However, a common requirement of games is to allocate a portion of memory *dynamically* at run-time, especially if the size of the block is not known at compile-time and it is required to persist outside of the current scope. Dynamic memory allocation is typically managed using a *heap*, which is usually a large block of free memory that is statically allocated for the game at load-time. The game does not necessarily need to know the organisation of the heap and instead relies upon an abstract interface to *allocate* and *free* memory blocks of a required size.

Given the limited constraints of many game platforms, memory management is a particularly important area of game code design. For this assignment, the aim is to implement an efficient dynamic memory manager within the framework of a very simple game example. The game itself need not be complete, but it must demonstrate multiple objects being created and destroyed inconsistently at run-time. The emphasis is on the heap itself, such as the methods to allocate and free blocks and how the lists of these blocks are stored. Particular attention can be given to efficiency, *fragmentation*, *memory leaks* and tracking.

If memory is allocated frequently during run-time, the memory manager must be able to quickly allocate a block of the required size. This requires an algorithm to rapidly search the free list for a block that is just the right size. Also, when blocks are allocated after previous ones have been freed, holes can appear in the heap where free blocks are dispersed among allocated ones. This fragmentation can result in there not being a free block large enough for a required allocation, even though the total amount of free memory should be enough to accommodate the request.

Allocating a piece of memory, but subsequently losing the reference or failing to clean up properly will result in a memory leak. These can be detected as the game exits and the memory manager can report on all remaining allocated blocks. However, it can be even more useful to track the file and line at which the allocation occurred, so that the bug can be found and fixed. This tracking information is also useful at run-time, where the memory manager could output a report of all currently allocated memory, including size and number of allocations from a particular file and line number. Finally, it may be useful to override the `new` and `delete` operators to use the allocate and free methods of the new memory manager.

Portfolio Project Submission

Prior to submission we recommend that you thoroughly review your Portfolio Project work for Section 2. You must pay particular attention to the presentation of your projects as this will be an area that we will be looking at during the assessment of your work.

Your submissions for the various projects will include text documents, 2D images, charts, levels, reports and other material. Any material you submit should be presented in a professional manner, having been spellchecked, play-tested and checked for consistency where necessary.

All paper-based submissions should be kept together in a suitable folder and any coding and assets submitted on optical media or memory sticks. Care should be taken to ensure that such media are package in such a way as to minimise the chance of damage during transit.

You should send your completed Portfolio Projects to:

Teaching Department
Train2Game
Freepost ANG7795
Luton
LU1 1BR