Fourth Generation Arcade Game Tools

Dr John R Rankin

Abstract – This paper investigates the extent to which it is possible for classical-style arcade games to be generated by an end-user using a suite of fourth generation tools. A game builder is a fourth generation tool that enables non-programmers to completely and quickly design and build an arcade game without requiring any programming. We find that the classical genre of arcade games share a commonality of features that can be encapsulated into a single sprite class. The game builder user designs, modifies and tests background scenes, sprite images and animation frames using various fourth generation tools and then invokes the game builder wizard that uses templates and the general sprite class and configures the template details according to simple end-user inputs to generate the final prototype. As a result a very wide range of arcade games can be quickly prototyped and tested by a non-programming end-user. Final versions of the game may need professional programming for fine-tuning for improved game efficiency. Also artificial intelligence in the computer game play may still require some specialist programming. Nevertheless the prototype shows off an end-user’s game idea to test for marketability. A cut-down version of the Game Builder program was readily implemented by a project team of final year Computer Science students.

Introduction

Arcade games have been with us since Toshiro Nishikado invented Space Invaders in 1978. There have been many versions of Space Invaders itself since that first program but it spawned a plethora of other outer-space shoot-em-ups, ranging from Asteroids to Galaxions and Space Zap [11]. Amongst this variety of arcade games the kind of user inputs and the game play or scenario are essentially the same (although the range of possible inputs has been enriched in the transition from video game boxes to PCs). Namely, for inputs one typically has a joy-stick to indicate direction and one or other fire and motion buttons. The player is represented by an icon or sprite (such as a spaceship) and this can be manoeuvred by these input devices to point in a desired direction, to move forward, to accelerate, to decelerate or to stop. The player can fire a range of weapons depending on the game and the firing is generally in the same direction as the player has selected for moving. Most frequently the weapon is a laser gun. This is where a straight line is drawn from the player to the first target hit. The laser fire hits the target instantaneously and is therefore easier to implement. A second sort of weapon is the bomb which is typically what the enemy sprites in the game fire at the player sprite. When a sprite emits a bomb it travels in an ordained direction and takes a finite time to reach its target. The bomb is therefore effectively another sprite moving over the background scene. A third possible type of weapon adopted in some games is the “photon torpedo” which is a sort of cross between laser and bomb. Like the laser beam, the photon torpedo fires in the direction of motion. Unlike the laser beam however it is not drawn as a single line from one sprite to the target but as a short line segment or dot which again unlike the laser beam travels at finite speed from the firing sprite to its target. Typically, photon torpedos are fired in bursts rather than single shots like bombs.

As the home and personal computer has evolved into a more powerful and window-based system arcade games have made the transition from dedicated arcade hardware to PC. The advantages of this transition have been discussed in [3]. The market for games and entertainment software for PCs has subsequently grown and flourished enormously [Compare references 1 & 2]. Games however have required expert programming skills especially to generate the fast smooth multi-object animations needed for arcade-style interactive video games. With fourth generation technologies [10] now breaking into the PC field it is now time for these technologies to be applied to the game and entertainment software area. Such fourth-generation tools would enable home users to create and tailor their own interactive arcade games to run on a windows PC. This research explores the possible scope for such fourth generation game builder tools. The level of difficulty in making such fourth generation tools is tested against final year Computer Science university student capabilities.

It is clear from this description of arcade games above that a suitably designed sprite class will cover the range of functionality for the games described above. This functionality is described in the next section.

Generic Sprite Class

Every graphic in the game that moves or changes shape over the background scene is regarded as a sprite, an object of the sprite class. A sprite class is therefore needed that will cover all the programming requirements for sprites in arcade games. We will look at this in terms of the data and methods required. Important sprite data includes the sprite
position, velocity, image, image size, number of frames, current frame number the movement limit box and state. The sprite is constrained to move in straight lines inside a prescribed box within the game window and flags are supplied to say whether the sprite disappears, bounces or renters from the opposite edge on hitting an edge for each edge of its movement limit box. This sprite data may be all public. They could alternatively be represented as class property set and get methods in more robust object-oriented programming. This protective style of object-oriented programming is commendable but however it is of dubious benefit in the unique area of games programming where performance is crucial. However, since our interest at this stage in making the Games Builder program is for games prototyping rather than end-product production, the best software engineering principles can nevertheless be applied.

We will now consider suitable public methods for the sprite class. The objectives in the sprite class design are firstly to remove all sprite coding from the applications main routines centralizing them into the sprite class as methods, and secondly to optimise the choice of methods to reduce the number that need to be called. The essential methods are the class constructor, initialisation and parameter setting routines, collision check, laser hit check, move, kill and revive methods. Since the class will contain components like images and image lists these must be properly constructed when each sprite object is constructed. The programming language IDE (Interactive Development Environment) can limit the form of the constructor(s) because of the need to incorporate these visual components and therefore initialisation methods are used to support every kind of appropriate parameter initialisation including setting the initial image, the animation sequence (if any), the initial position, velocity and movement limit box. The collision check method takes a circle as input and returns a flag saying whether this sprite’s hit circle overlaps the given circle or not. Circles are used for collision detection between sprites because the computation is fast and easy. The laser hit check method takes two points as input. The first is the start of the ray representing the laser beam and the second is the unit vector direction of the ray. The function returns a real number which if negative means that the laser beam did not hit this sprite. If the returned value is positive then the laser beam hit the sprite and the real value is the distance along the ray to this sprite. The game loop calls this method for all sprites against the current sprite in a loop and calls the kill method to set the sprite with the least positive return value to the HIT state. Finally the revive method is called for the sprite to its own particular initial values (which were saved internally by the initialisation methods). This is needed to save memory. For example a player can start with say three “lives” and therefore can be revived three times. Alien bombs are reused over and over and therefore are revived rather than allocating new memory for new bombs.

**Generic Arcade Skeleton & Student Projects**

All the development files needed to create a sample arcade game can be stored in a single directory. By analysing the game parameters—the image files needed, the number of enemy sprites, the types of weapons each sprite will use, the initial positions, velocities and limit boxes of each sprite and the keys that the player sprite responds to access various functions we can generalize the sample game to a generic game. The difference between the games created are only in the game parameters listed and these can be obtained from the user when he runs the Game Builder program and then saved as a single reference file. In this way the Game Builder does not need to create all the files involved in making the interactive game, and the generic arcade skeleton can be precoded and placed in a suitable directory ready to be tailored for a specific arcade game.

In setting student assignments we need to bear in mind the limited time for development available to students of around 30 hours. We also need to bear in mind the range of capabilities and programming strengths amongst students. The Game Builder concept is implementable by final year Computer Science students in groups of six or more with certain restrictions. The restrictions include that the games generated do not need sound, or introductory animation sequences, or high quality graphic artwork or different enemy alien types or more than one level of play or even game AI code. While any one of these enhancements is achievable by individual bright students the expectation cannot be given to the average student for the given limited project development deadline. General students have found this project easiest to do in Borland C++Builder. In contrast the brightest students can design graphics games with Visual C++, DirectX 8 or OpenGL but then these students would not want to work at that pace in a group. In the simplest implementation of the project, a windows project
asks the user for characteristic game parameters, such as scene file name, sprite image file names, number of enemies, initial positions and speeds, explosion animation sequence frames and keypress mappings. Finally the program saves these characteristics to a text file which is actually a C++ include file called GameData.h. The GameData.h file consists of C++ declared constants that the game skeleton recognizes and uses. For example, the key down event routine has a call to every possible player action option based on matching the key press to program constants in GameData.h. Many key press constants are mapped to the NO_KEY value so that the corresponding player sprite actions will not eventuate for the generated game.

The game skeleton also includes code for drawing and erasing the laser beams. In contrast to the bombs which as sprites use masked bitblitting with background capture and replacement for animation, laser gun and photon torpedo gun fire is drawn in the XOR mode. In response to the key press for laser fire, the on key down event sets a flag to say that the laser canon has fired. The laser beam is erased after a set number of game loop cycles after which refiring the laser canon is re-enabled. To simplify the student workload again, only one laser gun is provided in the skeleton and no phasors. It is assumed that the laser gun is for the player. However the player as well as the enemies can have bombs.

In the simplest implementation of this project the GB program writes the GameData.h file into a directory containing the game skeleton code. After this the user must compile all the code in the game directory and the end result is an executable windows arcade game. Improved versions would automate this step (so that the user has nothing to do but click a menu item) but this is beyond the student’s initial capabilities. Initially the idea of creating a fourth generation tool is frightening to even the best students because it is not yet within their scope of experience but I have seen that once the initial fear is overcome this project leads to great satisfaction amongst students and eagerness to make an improved version.

To enhance the games a program was made as a 4GL tool for generating random background space scenes. Options in the tool include generating various star shapes on the background and setting star colours. Constellation lines can also be generated randomly. Another two programs were made as 4GL tools to edit and test run bmp file lists as “anm” files. These anm files are to be used for small animations such as explosions.

**Future Work**

To convert the project from Borland C++ Builder to Microsoft Visual C++ takes a lot of additional laboratory classes because of the initial difficulties students find in learning Visual C++ and the drawbacks with automating the loading of images. However to convert the project from BCB Canvas graphics to Direct X under BCB is an easier path. The advantage of taking this path is that it results in faster and smoother graphics. Consequently the generated games are less like prototypes and more like end-products. This direction for improvement also makes the inclusion of sound effects easier by using DirectSound. After these improvements the appeal of the generated arcade games now depends more heavily on the quality of the graphics and Computer Science students are not renown for artistic skills so it is particularly unreasonable to expect them to produce that sort of quality within the short development time of an assignment. They are more adept at finding free images on the net. Finally the debate over DirectX versus OpenGL continues and a Game Builder based on OpenGL could as easily be designed. In particular, DirectX is less friendly to the two-dimensional graphics of classic arcade games than OpenGL or even DirectX 7. Our current student projects include web advertising of the Game Builder and free downloading but the project software doesn’t run on the web itself. So another direction for research is web-based Game Builder software for building web-based games [6].

The idea of a Game Builder program that will produce arcade games where the games start off easy but increase in difficulty and computer opponent plays with strategy is a direction for future research. Many authors [eg 4, 5, 7, 8] have worked on AI methods that enable the computer to grow smarter the longer it plays. Some authors have applied Fuzzy Logic [4, 9] and others have applied Genetic Algorithms [9]. This application area calls for a fast learning system or else specially supplied AI code. Further research directions include developing the sprite class into animated agents [8].

**Conclusions**

It has been proved possible to construct a fourth-generation Game Builder tool that will build classic two-dimensional arcade games like Space Invaders, Space Zap, Asteroids, Galaxians and many others by end-users who do not have to write any third-generation code at all. A simplified version of this Game Builder that generates arcade game prototypes has been successfully tested on final year Computer Science students as a group project. Tools designing background scenes and animating sprites have also been tested.
References