Technical Design

Technical overview of Lego Island 3
PC, PSX2 and Gamecube

Document Version 1.0.157

Author Phil Harris
Contributors Sam Swain
Matt Ritchie

Contents

1 Overview

This document details the technical design of the key areas of Lego Island 3 on PC, Playstation 2 and GameCube.

Much of the technical design will be based on the existing systems. However due to the new platforms, new engine requirements, a more advanced minimum PC specification and the new range of sub-games most of the game will be new code.

There are also several elements that were not present in Lego Island 2, primarily a more complicated island engine, a collectable card game and photograph and online elements.

1.1 Target Specifications

1.1.1 PC (Minimum)

Processor	
Memory	
Graphics	
Save Game Storage	Unlimited
Players	1-2 Split Screen
Internet Support	Yes

The PC graphics engine will be developed in a scalable manner allowing users with higher spec machines or better graphics cards to take advantage of more texture memory and more advanced features such as bump and environment mapping.

1.1.2 Playstation 2

Memory	32Mb DRAM
Graphics	4MB VRAM
Save Game Storage	8Mb
Players	1-2 Split Screen
Internet Support	No

1.1.3 GameCube

Memory	24Mb
Graphics	16Mb
Save Game Storage	512Kb
Players	1-4 Split Screen
Internet Support	No

Due to the limited save space available the photographs will be stored at a lower resolution on the GameCube version of the game.

2 Development Environment

2.1 Programming Language

Due to inconsistent support for C++ on the console platforms Lego Island 3 will be developed using C.

However, an object oriented approach will be taken to code design wherever possible. This will enable us to get some of the benefits of C++ even without the language support.

3 Graphics Engine

The graphics engine will be accessed through a standardised C interface but will be implemented specifically for each platform by the appropriate development team. On the PC the engine will use DirectX 8 as its basis. The consoles will use a custom developed engine and hardware specific libraries.

The consistent interface lets sub-games be written without reference to the target platform and allows the graphics engines on the consoles to be optimised to get the most out the available hardware. This will also make the sub-games relatively easy to port between the three platforms.

3.1 Game Styles

The graphics engine will support a number of styles of game.

3.1.1 Free Roaming

The free roaming engine is used for the main islands and sub-games played in large open areas. The player moves freely around the landscape and can interact with the characters and objects he finds there. Movement should be smooth with no interruptions and large areas of the map may be visible from some locations.

To this end, the entire map should be held in memory and multiple levels of detail used to increase draw distance by reducing the complexity of both objects on the map and the landscape itself.

3.1.2 Interior

Lego Island 3 will feature large and complicated interior locations, from palaces to underground dungeons to castles. Interior locations are made up of a number of “rooms”, again containing characters and objects for the player to interact with. Rooms may be very detailed and contain large numbers of objects.

Depending on the size and number of rooms the entire level may not fit in memory. A caching system will be required to load rooms as the player moves around the level. The PC version of this caching system can make use of any additional memory available to cache more of the level.

There are several ways of rendering interior levels ranging from simple models for each room with fade to black or animated transitions between those rooms, to Quake style portal systems where other rooms are visible through doorways etc. The system used will depend on the requirements of the interior levels and sub-games but will be kept as simple as possible to keep development time under control.

3.1.3 Restricted Movement

Many sub-games will take place in smaller, restricted environments. For example, within a single room, climbing up a tower, or throwing pizzas on a pitch. Although movement within these games may be fairly free ranging, the player is restricted to a relatively small area. Some sub-games may lock the camera and/or Pepper in position.

Because these environments are more limited, they can be more detailed than the larger island environments. Again, the entire level will be held in memory while the game is played.

3.1.4 Scrolling

Scrolling sub-games feature either horizontal and/or vertical scrolling or “in to screen” perspective scrolling. Levels for scrolling sub-games may be very large and require the levels to be streamed from disk as the game is played. However, careful design of the levels and reuse of graphical elements would mean that for some games the entire level could be held in memory.

3.1.5 Flight

Flying sub-games often require a large map, much of which is visible at the same time. As a result, a lower polygon model is needed. This is particularly true when flying over the islands.

However, by implementing a level of detail algorithm in the island engine the need for a special mesh can be avoided. As the player is often a long way away from the island it will be drawn in less detail anyway.

3.1.6 Cut Scenes

Cut scenes can take place on any of the levels. The cut scene script will be preloaded, however speech will be streamed. Some cut scenes may require more detailed models and these too will be loaded at the beginning of the cut scene.

3.2 Island Engine

In addition to a general engine used to render objects, buildings, sub-game environments etc. Lego Island 3 will require an engine for use on the islands in the game.

3.2.1 Map

There are several ways of representing the island map data.

3.2.1.1 Fixed Resolution Grid

The simplest representation of the map uses a fixed resolution grid. This makes certain operations much simpler, but provides an average solution to the problem.

3.2.1.2 Multi-Resolution Grid.

This system allows automated reduction of detail and would enable increased draw distance. It does however increase complexity of collision, shadow casting, etc.

3.2.1.3 Polygon

The map could be constructed out of an arbitrary set of polygons, or groups of polygons. This would give greater control over the landscape design, allowing the artists and designers to concentrate detail where it was needed.

Arbitrary polygon models present a problem, in that identifying applicable models to perform, say, a collision test on is difficult. This can be overcome with a block-based approach.

3.2.1.4 Block based

Both the above systems would work better in conjunction with a block based map structure. This would allow many optimisations to be built in for rendering and collision testing as blocks can easily be indexed and blocks to be considered are easily found.

3.2.1.5 Hybrid

All the above options have significant advantages and disadvantages, therefore, it seems that the best solution might be a hybrid consisting of a block based map with multi-resolution grids and polygon mesh models.

The models would be automatically broken up along block boundaries by the Maya export utility so that we can take advantage of the block-based system. The multi-resolution grids allow us to have an increased draw distance (consequently the models would need levels of detail built in).

3.2.2 Texturing

There are three main ways of implementing the texturing of the landscape:

3.2.2.1 Tiled

Each map cell can be mapped with any part of any of the map textures in use. This method makes it difficult to dynamically create levels of detail due to the complex texturing that you end up with. Other systems like deformable terrain also become much more difficult.

3.2.2.2 Paged

The visible parts of a single large texture are paged into a number of smaller texture pages on the graphics hardware. This gives greater control over the look of the level as every part of the map can be completely customised in graphical appearance.

This technique also allows other things such as deformable texturing, automatic texture generation tools, and variable level of detail algorithms. Possible problems with this are the memory overhead for storing such a large texture in system memory, and the processing cost of uploading textures as you move around the map.

3.2.2.3 Mixing

With this approach, several different tileable textures are mixed onto map polygons using transparency. This technique works particularly well with automatic generation of map textures, to the extent of being able to alter the texture on deformed landscape in real time.

The down side is that you get much less control over the actual texturing. This system also doesn't allow automatic detailing.

With paged and mixing systems described above, there is great scope for enhanced tools to be made available. For example you can start with plain grass all over the map, then build up other textures/images using automatic tools, hand painting/spraying, and manual placement.

The paged system best meets the needs of the other graphics system considerations such as; map level of detail and arbitrary deformable terrain. However, the limitations of the consoles may mean that a different solution will need to be used.

3.2.3 Collision

Many operations require collision tests with the landscape and constituent models. This is traditionally a time consuming operation, but with the block based map system detailed above, it will already be a more optimal solution.

3.2.4 Level of Detail System

An automatic level of detail system for the landscape engine would eliminate fogging on many systems, including the consoles. Using this system, landscape polygons that are further away

from the camera and small on-screen are combined with nearby polygons to reduce the amount being drawn.

Using a level of detail system does limit the approaches available for texturing, in particular tiled textures do not work well with a level of detail system.

3.2.5 Deformable Terrain

The design for Lego Island III includes deformable terrain. Two approaches are available for this swapping map data or arbitrary deformable terrain.

3.2.5.1 Swapping Map Data

Using this approach one area of map is replaced with another entirely different piece, with some sort of transition for dramatic effect. For example, a collapsing building might sink slowly into the ground while a particle effect plays around its base. Once the building reaches ground level the model is swapped for a model of the destroyed building. The particle effect hides the change.

The advantage of this method is that it places no particular restrictions on texturing, the two models can be textured independently using any of the above techniques. However, this approach does limit deformable landscape to pre-scripted areas and objects.

3.2.5.2 Arbitrary Deformations

Using this approach the landscape is modified directly in response to the player's actions.

For example, a player might drop an anvil from the top of a building that would land on the ground and leave a "dent".

Depending on the texturing technique used the texture could also be changed to reflect the damaged ground. The amount of deformation is limited to ensure the player doesn't create huge holes they can't get out of etc. A more advanced system would allow buildings to be "dented" by projectiles.

Arbitrary deformations place restrictions on the type of texturing used in the game, tile based texturing is unlikely to work well when the landscape is deformed.

Depending on the game design, either or both of the above techniques could be used. Arbitrary deformations would be used on specific areas of the landscape.

For example, a sandy beach would be marked as deformable in Maya and could then be modified by the player. A tide could then wash over the beach and remove the deformations, flattening the landscape back out.

3.3 Lighting

3.3.1 Static Lighting

A number of types of lights can be implemented and made available for lighting levels and sub-games.

- Directional/infinite lights
- Point lights.
- Spotlights.

- Dark lights, removes light from the level for tweaking lighting levels.
- Offline bump mapping using a depth map provided by artists.

These lights can be used to create light maps or vertex based lighting for the levels.

3.3.1.1 Vertex Lighting

Vertex lighting changes the polygons on the landscape based on the current lighting levels. These values can be pre-calculated but can also change dynamically allowing day and night transitions and other similar effects. Depending on the resolution of the graphic data vertex lighting generally looks worse than a light map.

3.3.1.2 Light Maps

For shadows caused by the shape of the landscape and landscape objects that don't change such as fence posts or bridges, the shadow can be rendered onto a light map. This provides a very realistic fuzzy shadow but depending on the graphics hardware may require additional processing. Shadows for moving objects must still be generated using one of the methods detailed below and day and night transitions are much harder.

3.3.1.3 Texture Maps

If a single texture is used for the entire level, shadows of static objects can be rendered onto the texture itself. However, this technique is less useful if transitions from day to night are required and shadows for moving objects must still be generated using one of the methods below.

3.3.2 Dynamic Lighting

A number of possibilities exist for dynamic lighting.

3.3.2.1 Distance Based Mapped Texture

Using this technique a texture is mapped onto the landscape to represent the light as it hits the ground and buildings.

3.3.2.2 Hardware Lighting

Some cards support hardware lighting to provide fast diffuse, specular, and bump mapped lighting effects.

3.3.2.3 Spot Lights

A simulated beam effect with a dynamic lighting effect at the end. Useful for searchlights, stage lighting, car lights etc.

3.3.2.4 Halo Lights (Small Glows)

Useful for giving the appearance of lots of small lights at a low rendering cost. For example, runway lights, fairy lights, fireflies or spooky eyes peering out of the darkness.

3.3.2.5 Lens Flare

This simulates the lens flare that sometimes occurs in films and can be useful for giving a film like feel to certain areas of the game.

3.3.3 Shadows

Several types of shadows are available for characters, providing varying degrees of realism and processor cost.

3.3.3.1 Fuzzy Blobs

These provide a very simple and perhaps more cartoony shadows by projecting a soft-edged elliptical texture onto the area surrounding the characters. Single shadows of this type are only really good for models in the distance where just the suggestion of a shadow is all that is required.

More realistic shadows can be achieved by projecting multiple blobs from attachment points on the models, e.g. arms, forearms, torso, head, thighs, and legs. Where these overlap on the ground they build up a reasonably realistic soft shadow.

3.3.3.2 Rendered Shadows

These are created by rendering a model onto a texture in black from the point of view of the light, then flat projecting this onto the surroundings. This can be costly on some cards that don't allow texture rendering but it can still be done. This produces probably the most realistic soft shadows but they can appear a little jagged.

3.3.3.3 Stencil Buffer

Using the stencil buffer capabilities of a lot of modern graphics cards, this would provide very realistic shadows. They could either be projected from a lower resolution version of the actual model or from a special flat shadow model pre-generated from the actual model. The model casting technique is more realistic but computationally more intensive. This may not be a problem if only a few of the near-by characters have them. Stencil buffer shadows have very hard edges that may or may not be desirable.

3.4 Animation

There are a number of animation systems available, each with it advantages and its disadvantages. The three most common are as follows:

3.4.1 Vertex

Vertex animation presents each frame as a complete new vertex set for the polygon data to reference. This causes the animations to become quite large but does allow complete control when animating.

3.4.2 Hierarchical

Hierarchical animation is stored as sets of function curves for rotation, scaling and translation values for each child component in a hierarchical model. This type of animation is good for mechanical models where rigid pieces are moved relative to each other (e.g. a tank turret).

3.4.3 Skinning

Here, an invisible skeleton (as a hierarchically animated model) is animated beneath the actual polygon skin. Each vertex is controlled by a weighted contribution from one or more of the 'bones'.

This is good for realistic flexing of a character's skin and provides a more efficient method of storing animation data. It does however usually require a high amount of computation to implement.

3.4.4 Summary

A summary of these animation systems is given in this table:

Type	Storage	CPU	Gfx B.W.	Tweenability
Vertex	High	Low	High	Low
Hierarchical	Mid	Mid	Low	High
Skinning	Low	High	Low	High

KEY: Gfx B.W. relates to how fast data typically needs to be 'pumped' to the graphics card to perform the animation. Tweenability is how easy it is to automatically tween or interpolate frames of animation.

For maximum flexibility and efficiency all the above and combinations thereof will probably need to be supported.

3.5 Effects

The exact design of the game will determine what effects will be developed, however a number of options are available.

3.5.1 Particle Effects

The particle effects system will be a combination of the best parts from the current systems on Lego Island 2 and Dogs of War. The three platforms may require a slightly different approach to particle effects to take advantage of the particular strengths of the system. With this in mind, the particle system will be designed with a standard interface and implemented on each platform independently in the same way as the graphics engine. All platforms will support the same functionality.

A graphical particle system editor will be available on the PC that shows exactly how particles will appear in-game. This will allow high quality particle systems to be created very easily.

Depending on the requirements of the game design a number of features will be considered for inclusion in the particle system.

- Optional particle collision, for example bricks bouncing around and off of each other.
- Global wind direction and speed so that smoke is blown across the map in a consistent way etc.
- Control functions for particles to allow custom effects to be programmed. For example, fluttering leaves.
- Volumetric particle systems for creating banks of smoke, dust and fog.
- Particle texture page split with half for standard particle designs, and half for sub-game/island specific designs. This will prevent different people from "breaking" other people's effects.
- Ideally the texture page for effects will be built dynamically based on the systems currently running to allow greater variation of the effects available.

3.5.2 Water

Being on an island, a system for providing water would be required. The Dogs of War water system will be used as a starting point.

The following enhancements and additions will be considered:

- Shoreline waves that lap gently (or not!) on the beach.
- Running water in rivers, perhaps following the flow of the river correctly.
- Switching to a detailed rippling version of the water when you get close to it. This would mean that as a character moves through water or raindrops hit the surface, the water ripples and reacts appropriately..
- Variable styles of water from perfectly calm lakes to stormy seas.
- Blurred reflections would enhance the appearance of the surface, particularly as you looked across from one island to another.

3.5.3 Sky

Again, the Dogs of War layered sky system will be used as a starting point and the following improvements considered:

- Sky objects, e.g. sun, moon, stars, sunsets.
- The detail of the textures used for the sky may be increased.
- Possibly more passes to create more layers of sky for more subtle control.
- The ability to “morph” between different skies, to allow weather/time-of-day transitions.

3.5.4 Night and Day

The Lego Island 3 design makes use of day and night transitions and the graphics engine will need to support the changing light levels, shadows etc. This will have a significant effect on the way lighting is done, on the main islands in particular. Vertex lighting is the most suitable way of supporting these transitions and with a sufficiently detailed island mesh should give acceptable results.

3.5.5 Weather

Weather may also play an important part in the game. A number of possibilities exist, including.

- Sunny skies, with little fluffy clouds.
- Rain with a rainbow, thunder & cute lightning. Ground that has been deformed by the player could fill with water as it rains.
- Snow settling on the ground and buildings. As characters move around they could leave footprints in the snow.

Weather effects open up a number of design possibilities including allowing the player to control the weather either globally, or locally by “flying” a rain or snow cloud around the landscape.

3.5.6 Effects Layer

By using a single large map texture light maps can be incorporated directly, rather than with a second texture operation. This frees the second texture pass up for other things. Similarly using vertex lighting also leaves a texture pass free for other things.

Possibilities include:

- A cloud layer. The second layer could be used to cast moving cloud shadows across the landscape and buildings.
- Snow layer. During snowy weather the ground could slowly cover with snow. Characters could leave footprints in the snow as they walk.
- Reflectivity mask by fading out the map to reveal reflected map/models/sky to create the appearance of puddles.

3.6 Scaleability

Although the consoles have a standard configuration, PCs vary wildly in power. Therefore, Lego Island 3 will support a number of options to ensure that the game runs well on a variety of systems. These options also allow more complicated scenes to be rendered by reducing the detail of less important elements.

3.6.1 Level of Detail

Characters and buildings within Lego Island 3 will support multiple levels of detail (LOD).

Using this approach, models that are further away from the user and smaller use a much less detailed model. This can dramatically reduce the number of polygons being displayed at any one time and enables more complicated environments to be created. Levels of detail can also be used to keep frame rates consistent by dynamically reducing detail when a lot of models appear on screen at one time, allowing crowd scenes etc.

Using models with multiple levels of detail also builds in a degree of scaleability. Less powerful systems can use a lower detail model as their top level of detail to reduce the number of polygons in a typical scene. This becomes essential when playing multi-player games in split screen mode. Similarly it allows more detailed models to be available in the PC version of the game to provide a certain amount of future proofing.

Two methods can be used for implementing levels of detail, both of which will be considered for Lego Island 3.

3.6.1.1 Multiple Models

The simple technique uses a number of models, one for each level of detail. The models are created either by hand, which is time consuming but generally produces very good results, or using an automated system. These models are switched in and out based on distance from the camera or size of the model on screen. By creating the level of detail models carefully and choosing appropriate distances for switching between them the level of detail system works virtually invisibly with little or no “popping” of buildings and characters as levels of detail change.

3.6.1.2 Variable Detail

This technique uses a high-resolution model for the top level of detail and then dynamically reduces the complexity of the model as it gets smaller. This is a more complicated system and requires more processing power, but is more flexible than using multiple models. In particular it does not require multiple sets of data for each model and the work that involves.

DirectX 8 provides an automatic system for variable level of detail that may be suitable for use on the PC depending on the quality of the results. However, GameCube and

Playstation 2 will need their own systems and a consistent approach across the three platforms may be more appropriate.

3.6.2 Detail Control

Many graphical options have various levels of complexity. For example, model detail, type of shadows etc. Detail settings will allow these options to be controlled so for example the player can use simpler shadows but higher resolution character models.

3.6.3 Automatic Performance Levelling

By monitoring the current frame rate the game can automatically adjust the detail in the current scene, so for example when a large number of characters appear on screen the engine may start to use lower resolution models to keep the frame rate at a suitable level.

3.7 Multiple Viewports

In order to support multi-player games on a single system the graphics engine will provide support for multiple viewports on both PC and console platforms.

Any number of viewports can be defined each with a position and size. Graphics output is scaled and clipped to the currently selected viewport.

For multiplayer games, two, three or four viewports are defined and the game state for each player used to draw the scene in each viewport. Reduced levels of detail are used to ensure that the frame rate in each viewport stays at a suitable level.

Generally the single player game uses a single viewport covering the whole screen. However, there are uses for multiple viewports even in single player mode. For example, an onscreen map, a small scanner showing a modified version of the view from the player's current position, cameras moving across the game world or dual monitor support on the relevant NVidia and Matrox graphics cards.

3.8 Diagnostics primitives

A useful feature for debugging is the ability to easily display diagnostics graphics primitives. These enable programmers to display additional information that will not appear in a release version of the game. For example, a display of the route a character is currently taking, health levels or the current value of internal variables.

The following primitives would typically be most useful:

- 2D and 3D Line with optional arrowhead for vectors.
- Text.
- Point marker.
- Polygons, three or 4 sided.
- An icon from a standard diagnostics texture page.

3.9 FMV Playback

Any FMV required by the game will use MPEG compression. The PC version will use DirectX 8 to playback the movies. The consoles will use their own hardware decompression.

3.10 Advanced Graphics Features

A number of advanced technologies are available which could be used to enhance the game engine depending on design requirements. Support for these features varies between platform and on PC from system to system so none of them will be required by gameplay.

3.10.1 Hardware Transformation and Lighting (T&L)

Hardware transform and lighting is supported by a number of modern PC graphics cards and provides a significant speed increase when rendering models. The PC graphics engine will be designed to take advantage of this facility if it is available.

3.10.2 Full Screen Anti-Aliasing

Full screen anti-aliasing softens the edges of graphics, removing jagged edges. Anti-aliasing does incur a speed penalty but is relatively easy to add as an option for cards and systems that support it.

3.10.3 Motion Blur

On most systems there is a heavy cost for motion blur (the 3DFx T-Buffer does reduce this cost on the PC). Therefore, this would need to be limited to special occasions (perhaps during a few cut scenes).

3.10.4 Depth of Field

Similarly depth of field is usually very processor intensive. Again this could be available as an option for cut scenes.

3.10.5 Bump Mapping

Simulates surface bumps with specular lighting. To get the most from bump mapping light sources need to move. This can produce some spectacular looking results under the right circumstances, bumpy ground, textured cloth, grass etc.

3.10.6 Cubic Environment Mapping

Cubic environment mapping produces realistic reflections by mapping the surrounding area onto an object. For example, a knight's suit of armour would reflect the scenery around it. However, this is very processor intensive process and is really only suitable for controlled situations. For example, a specific sub-game where Pepper is standing in the centre of an environment wearing a reflective suit and does not move around much.

3.10.7 Per Pixel Effects (Lighting, etc)

Per pixel effects are used to create higher quality lighting and other more complex features. Implementing them is a fairly complicated process and support is very limited so these are unlikely to be of much value to this project.

4 AI

4.1 Behaviours

The basic design of the AI systems will be similar to that in Lego Island 2.

Every game object can have AI associated with it in the form of a chain of ‘behaviours’.

Behaviours typically include:

- Automation of objects. Lifts moving, doors opening etc.
- Mini-Figs walking about using the waypoint map.
- The individual reactions of characters in the game. For example, pointing where Pepper should go, reacting to the weather etc.
- The movement and behaviour of the Brickster Bots.
- All of the special behaviours in the sub-games.

Each frame the current behaviour is used to update the state of the game object and optionally switch to a different behaviour. By chaining multiple behaviours together, quite complex and seemingly intelligent behaviour can be attached to any number of characters relatively easily.

For example, the basic walking behaviour might detect when it is raining and change to a wet weather behaviour that may be unique for each character.

One of the advantages to this approach is that it allows behaviours to be reused where appropriate to produce consistent and reliable behaviour throughout the game.

Parameters associated with the behaviours (speed, rate of fire, how concerned a character is about getting wet etc.), will be editable without needing to recompile the game so designers will be able to tweak playability very quickly, without the help of the programmers.

4.2 Character Behaviour

In addition to plot driven behaviour, characters could also ‘live’ their own lives as the game progresses, giving plenty of opportunity for interaction with the player.

For example, a character may work in their garden for a while before walking down to a shop, buying a few groceries and heading home carrying a pair of heavy bags. If he happened to be passing, Pepper could help by carrying the bags home and be rewarded with objects, game cards etc.

These behaviours can be made as simple or complicated as required and can vary throughout the game. Wednesdays might see Nick Brick make his regular report on crime on Lego Island. If Pepper goes to the police station at the appropriate time he’ll find a lot of the inhabitants of the island listening to Nick explaining that yet again it has been a very quiet week on Lego Island, crime wise.

4.3 Incidental Creatures

Similarly, to make Lego Island an interesting and dynamic environment the island could be populated with a number of incidental creatures for example birds, fish and rabbits. These creatures would “live” very basic lives on the island, rabbits will graze on hillsides, birds roost in trees and circle around the island etc. Their behaviour could also change as the game progresses.

For example, at the beginning of the game a couple of rabbits are out grazing on a hillside. Pepper walks over the hill and into view of the rabbits that then run back to their burrow. A while later, assuming Pepper has left, they will return to graze. However, if they “see” Pepper several times they will become more and more tame until eventually they will not run away from Pepper and he can get close to them, perhaps unlocking the secret to a sub-plot.

5 Physics

A simple physics system that uses bouncing, inertia effects will add significantly to the game world, particularly if those effects were exaggerated to produce a cartoon like effect.

By using AI behaviours to create a general physics system that implements a number of rules about the game world, a great deal of interaction and interest can be included in the game without having to code each case.

For example, bumping into a tree would cause it to wobble, which would displace an apple which would drop from the tree and bounce and roll down a hill before coming to rest against a fence post. Similarly a rock could be pushed down the hill until it hit the fence post, perhaps breaking it and annoying the owner of the fence.

6 Game Management

Lego Island 3 maintains the sub-game based structure of Lego Island 2 and it is useful to treat all areas of the game from the initial FMV to the front end to the sub-games and islands the same way. By setting a standard interface for setting up and running all sub-games, they can be automatically managed by the front-end.

6.1 The Game Manager

The Game Manager library is the glue that holds all the sub-games together. It is responsible for starting and stopping sub-games at the appropriate times and for updating AI behaviours and the sub-games each frame.

6.2 Game Definition

Each sub-game has a game definition associated with it. This definition specifies a game identifier, game object list, the address of initialisation and update functions, game title etc. Miscellaneous information such as author name, revision number and revision date, can also be stored in this structure and updated by the sub-game editor.

6.2.1 Game Identifier

Each game has a numerical identifier associated with it. The Game Manager uses these unique identifiers to switch between sub-games during the game.

Certain game identifiers are reserved and have a special meaning.

- GID_NONE represents the initial or error state, where no sub-game is currently running.
- GID_FIRST moves straight to the first sub-game in the list.
- GID_NEXT moves on to the next sub-game in the game list. If no more games exist the game manager switches to GID_NONE.
- GID_PREVIOUS moves back to the previous game in the game list. If the current game is the first in the list the game manager switches to GID_NONE.

6.2.2 Game Object List

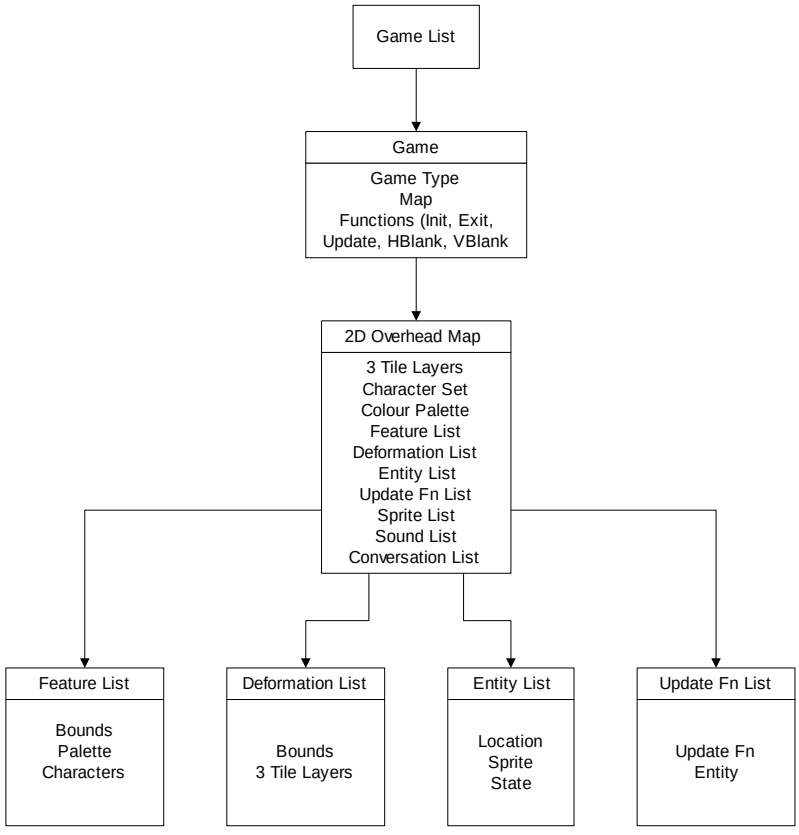
The Game Manager uses the list of current game objects to update the behaviours each frame.

6.2.3 Functions

Each sub-game has three functions defined, one for initialisation, one for frame by frame updates and one for shutdown. The Game Manager automatically calls these functions at the appropriate time.

6.2.4 Game Title

The game title is a simple text string and is used to refer to the game in debug menus and messages.



6.3 Game List

The game list is present in every build of the game and is used to specify which sub-games are available. The list will be edited by hand or generated automatically by the compilation process and is simply a list of game definitions that have been included in the current build

For example,

```
const GAMEDEF *g_GameList[]=
{
      &BrickDive_GameDef,      // Brick Dive sub-game definition
      &WhackABot_GameDef       // Whack-A-Bot sub-game definition
};
```

The Game Manager begins by starting the first sub-game in the list. The sub-games themselves are then responsible for controlling progress through the game.

When a sub-game is complete the Game Manage can be switched to another sub-game in one of two ways.

6.3.1 Switching to a New Sub-Game

Sub-games can switch completely to a new sub-game using the `GotoGame` function.

```
void GotoGame(u32 GameID);
```

When the current update is completed, the Game Manager will switch to the game with the supplied ID.

6.3.2 Branching to a New Sub-Game

Alternatively a sub-game can switch temporarily to a new sub-game using `GosubGame`.

```
void GosubGame(u32 GameID);
```

When the current update is completed, the Game Manager will switch to the supplied game just as before. However, the current game is ‘pushed’ onto a return stack. The new sub-game can ‘return’ at any time using the `ReturnGame` function:

```
void ReturnGame();
```

When this function is called control returns to the calling sub-game as soon as the current update is complete. If a sub-game calls `ReturnGame` when there are no sub-games waiting on the return stack the Game Manager will instead move on to the next sub-game in the list.

A sub-game called using `GosubGame` can call other sub-games with `GotoGame` or `GosubGame`.

6.4 Game Manager Example

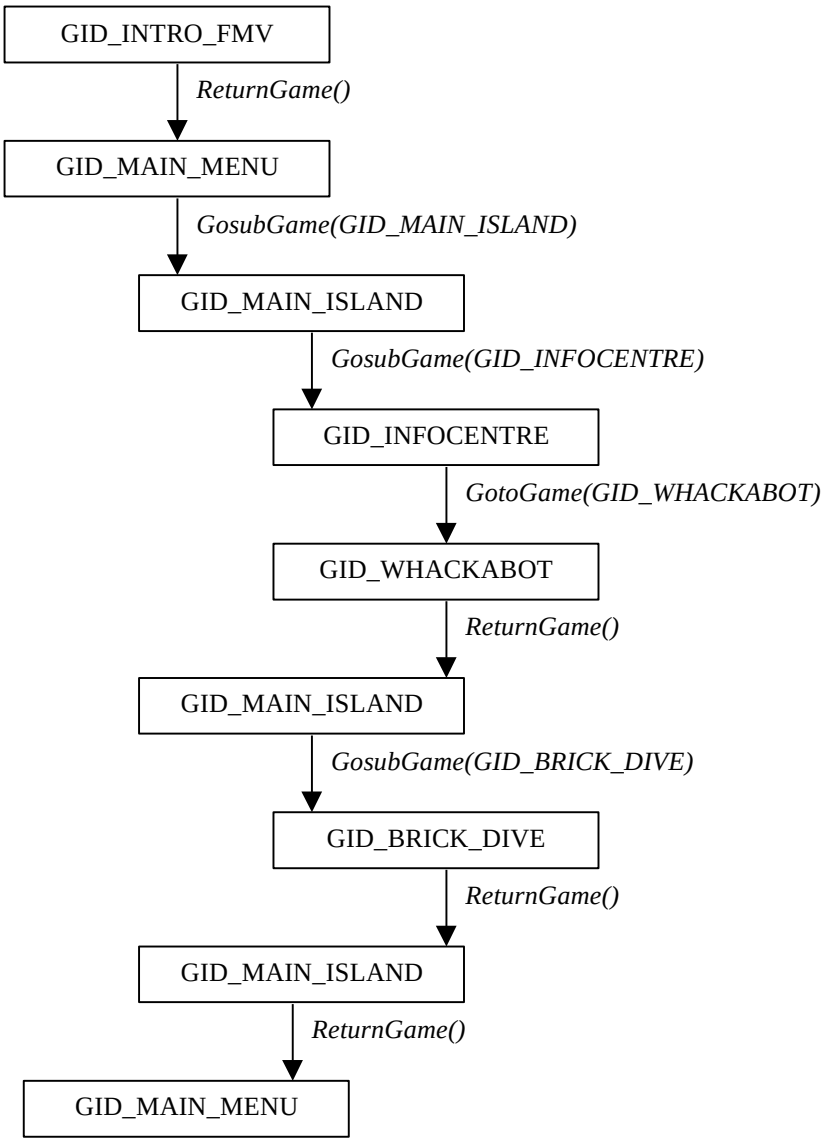
The current build of the game contains six sub-games, defined in the following order:

GID_INTRO_FMV
GID_MAIN_MENU
GID_MAIN_ISLAND
GID_INFOCENTRE
GID_BRICK_DIVE
GID_WHACKABOT

The Game Manager might run the sequence as follows:

Game manager begins with first sub-game.

Sub-game calls ReturnGame() with, no game on the call stack so the Game Manager moves on to the next game.



7 Audio

7.1 PC

Lego Island 3 will use the Miles Sound System used on Lego Island 2 for sound effects and speech. Miles provides a simple, consistent and reliable interface to a number of types of sound including mono, stereo, 3D and Dolby Surround Sound.

Creative Labs EAX standard will be supported which allows reverb and echo effects to be added. These effects will be “painted” onto the game maps and used to modify sounds dynamically.

So for example, as Pepper walks underground from outside the sound of his footsteps will change, echoing around the player as he walks through the tunnel. More advanced effects are also possible, for example sounds from the next room become muffled when the door to that room closes.

Speech will be compressed using MP3 compression and will be streamed from disk or CD as required.

Music will be implemented through DirectMusic and will use the system’s interactive capabilities to provide a dramatic soundtrack to the game.

7.2 GameCube and Playstation 2

Both consoles will have a custom sound engine written for them. The interactive music system will not be implemented.

8 Collision

Collision will again use the general-purpose system used by Lego Island 2.

The collision library is available to any sub-game and contains a number of functions used to check for collisions between lines and objects in the game world and the landscape. Sub-games and behaviours use these functions as they see fit.

9 Cut Scenes

Cut scenes will again form a strong part of the game. A graphical editor will be developed to allow artists and designers to quickly combine speech, camera movements, character animation and special effects into cut scenes. The editor will include real time feedback using the in-game engine making it very easy to create high quality cut scenes.

The final output of the cut scene editor will be a text based script file that is loaded at the beginning of the cut scene. Some cut scenes may make use of higher detail models for close ups etc. and these too will be loaded at the beginning of the cut scene. Speech will not fit in memory so will be streamed as required.

9.1 The Cut Scene Editor

This system will behave like the animation systems used in many 3D animation packages, and shares some of the concepts, most notably those of function curves and key frames. The system will however take the idea further to include integration with the graphics engine and management systems used within the game. This allows greater scope for control of parameters

and triggering of events without the integration complexity and slower feedback of using a general purpose third party editor.

A cut-scene editor allows sequences of events and changes in parameters to be choreographed to occur over time. Interactive editing allows better freedom of expression and realism to be incorporated in these sequences. In-game cut scenes provide a much more immersive experience than, say, pre-rendered FMV sequences as they take place in the same 'world' as the game itself. Other advantages are that they will take up less space, they are more flexible and versatile, and can be more interactive.

9.2 Playback

Multiple sequences can be triggered concurrently and if required associated with a graphics engine viewport (for camera control or screen effects).

Sequences can trigger and control other sequences and can be used for a wide range of things from controlling a simple animating object to running complicated cut-scenes.

9.3 Channels

User controllable values are available as channels in the editor. Each channel has key-framed function curves or trigger points set-up on a time-line. Some channels can be edited indirectly. For example, moving a camera or entity in the game world will modify the curves of the appropriate channels in the editor.

9.3.1 Channel Types

Two types of channels are available.

9.3.1.1 Analogue

Typically value parameters that change in a continuous manner, e.g. position.

9.3.1.2 Digital

Event triggers or states. These can have more than just two binary states.

9.3.2 Uses of Channels

Examples of some of the channels that could be available:

- Camera parameters. Position, focal point, field of view, special effects etc.
- Entity parameters. Position, animation, colours, etc.
- Special Effects. Screen space effects, distortion, filtering, etc.
- Event triggers. Spawn entity, deform landscape, swap object etc.
- Particle system triggers.
- Interface triggers and control. For player tutorials.

9.4 Flow Control

9.4.1 Time

The progression of a sequence will usually be linked to real-time from the point it is started, but may be controlled by the function curve of another sequence.

9.4.2 Loop, Repeat, Skip

Loop, repeat, and skip points will be available to provide more powerful sequencing.

9.4.3 Wait

Wait events can be added to pause the sequence progression until triggered by another event. For example a key press, or waiting for a character to finish walking to a certain location.

9.4.4 Sub-Sequences

Sequences can be used to trigger (and control) other sequences to provide a procedural style control mechanism.

9.5 Interfaces

The channels available from everything that is controllable are exposed via a standard query mechanism. The editor can therefore keep up to date with the channels available automatically as they are added to the different parts of the system.

9.6 Re-use

The editing and playback systems can be written with re-use in mind and by making the systems general purpose they can be re-used in a number of projects.

10 Player Control

Although control systems will be kept as similar as possible across the platforms, there are significant differences in the controllers, particularly on the PC where the keyboard and mouse may be the only options available.

To get around this problem, the player's control input will be presented to sub-games in abstracted form. In cases where the consoles support features not available on the PC (using the analogue control for varying the speed of Pepper's walk for example) the PC control system will mimic the operation of the consoles (by providing a sneak key perhaps).

This approach also allows controls to be redefined without affecting the sub-games.

The control system will also handle multiple controllers, the sub-game need only ask for the controls relating to the player they are currently updating. The control system will handle where that controller is and whether it is a joypad, keyboard, joystick or mouse.

11 Trinkets

Throughout the game the player can collect artefacts found all over Lego Island. These items may be found lying around or earned as rewards when games are completed. Trinkets are not required to complete the game, they are simply another reward.

The items available will be determined randomly and some trinkets will be rarer than others. To enable the player to collect the complete set they will be able to trade with friends, either over the Internet (PC only) or via memory cards on the consoles.

A trading post will also be provided where the player can swap trinkets with in-game characters.

12 Trading Card Game

Lego Island 3 will include a collectable card game, similar to games such as Pokemon and Magic: The Gathering.

At the beginning of the game Pepper will be given a randomly generated deck of cards and can challenge other characters within the game world or play multiplayer against a friend. Victory over your opponent wins you a card. Other cards can be found or are given as rewards during the game. The player can also trade the cards with friends or with characters in the game.

Each character within the game will have a specific skill level and style of play and have a number of decks that they will use to play against Pepper. Early characters may only have one or two decks but later on, more difficult opponents may have a large number of decks to keep Pepper on his toes.

The card game will be implemented as a sub-game and will be accessible by approaching any character in the game. Once defeated, a character can also be challenged from the front end.

12.1 AI

The AI for the trading card game will have a number of parameters used to control how a game is played. By varying these parameters and providing different decks each character will provide a different challenge and allow us to gently increase the difficulty of the opponents in the game.

12.1.1 Attack

Dictates how good the character will be at creating attacking combinations of cards. A character with a high attack rating will create very strong attacks to weaken the player.

12.1.2 Defence

Dictates how well the character will create defensive combinations of cards. A character with a high defence rating will be very good at creating impenetrable defences and blocking the player's attacks.

12.1.3 Aggressiveness

Controls how likely the character is to choose an attacking move. Characters with a high aggressiveness will prefer to create and use attacking combinations of moves while lower rated characters will concentrate on building defences. The quality of the character's defences and attacks will depend on their Attack/Defence rating.

Aggressive characters are also more likely to respond to threats by removing the threat from play rather than creating a defence to block it.

Less aggressive characters will be slightly slower at finishing off their opponent giving the player a chance to recover and will favour repairing damage done to them over damaging their opponent.

12.1.4 Reflexes

This controls how good the character is at responding to the attacks from the player. A character with high reflexes is more likely to block an attack by the player using a card in their 'hand' or in play.

12.1.5 Observation

A character with high observation skills is more likely to respond to cards played by Pepper. For example, if the player puts a strong attacking combination into play a character with high observation will be more likely to attempt to play a card to block or eliminate the threat.

12.1.6 Caution

Caution determines how likely the character is to play a risky combination of cards. If a particular combination relies on a rare card appearing, a less cautious character will start creating that combination in the hope that the correct cards will appear, cautious characters will favour combinations they already have. They will also be reluctant to begin creating attacks unless they have a defence in place.

12.1.7 Intelligence

Intelligence controls how likely the character is to pick a good card to play at any time. A character with a high intelligence will always play the best card they can based on their other preferences. Characters with low intelligence might pick a much less effective card.

A character with a high intelligence is also more likely to 'remember' which cards have been played by either player during the game and use that information to make decisions. For example, if the character has three cannon balls in its deck and none have been dealt yet even though there are only a small number of cards left to play, an intelligent character will be more likely to prepare a cannon for when they appear.

12.2 Decks

Predefined decks will be built for the game and characters will be given a number of decks to choose from. Each deck will have variants that use stronger cards, later characters will have these stronger decks made available to them.

The difficulty level for a character can be controlled using the decks available to that character. Characters may also base their choice of deck on the cards the player has available to either make it easier or harder for the player to win.

13 Multiplayer

Many of the sub-games will be designed to support multiplayer and some multiplayer games may also take place on the main islands.

13.1 Split Screen Multiplayer

Multiplayer sub-games will be played using a split screen, up to four players on GameCube and two on Playstation and PC.

Split screen mode will be implemented using the viewport system in the graphics engine with the player control library taking care of the complexities of multiple controllers.

Every sub-game will be written to support multiple viewports and controllers. However, the Game Manager will also implement as much multiplayer functionality as possible and depending on the design a sub-game may not need to know that it is being played in multiplayer mode.

13.2 Online Features

Neither of the consoles will feature any kind of online play. Multi-player games will be limited to split screen only.

The PC version of the game will include Internet support. However, this will not include the ability to play multiplayer games over the Internet. Instead the online support will take the form of Internet trading of trinkets and collectable trading cards.

13.3 Trading

13.3.1 Consoles

Trading on the consoles will be possible by memory card only, not via link up cable. Visiting a “trading post” within the game will check any memory cards present for saved Lego Island 2 games. Each game present will appear as a “shop” in the game which the player can visit to perform trades.

A security system of some kind may be needed to try to ensure that the player who owns the memory card approves of the exchange. For example, a password made up of control pad buttons may be required before the swap can take place.

13.3.2 PC

Trading on the PC takes place by logging onto a central Lego server. Each time the player logs onto the server their current inventory is uploaded for everyone to see.

While logged on, players can search for other players who have specific items they are looking for and make an offer. The offer takes the form of a message which arrives in the other player’s “mailbox” next time they log on showing who is making the offer and what they would like to trade. The offer can either be accepted or rejected with a single keypress.

14 In-Game Photography

Due to the complexity of the environment in the game, Pepper’s camera will capture an actual screenshot in bitmap form. Because of the limitations of memory card space on the consoles these images will be reduced in size and stored in compressed format, possibly JPEG compression.

15 Memory Management

Memory management will use the same system used in Lego Island 2.

Memory is obtained from the system in a number of large blocks that are then allocated to the game by a custom memory manager.

Memory allocation is managed on several levels.

- Permanent memory is allocated once and never released.
- Level memory is allocated for the duration that a level is in play.

- Sub-game memory is allocated for the duration of a sub-game.
- Frame memory is allocated for the duration of a frame and should be used very rarely.
- Disposable memory is allocated for the duration of a function or process.

15.1 Resource Tracking

At any time, the memory manager can be asked to log information about `alloc` and `free` pairs identifying any lost memory chunks and preventing reuse of lost pointers.

15.2 Strong Memory Block Protection

Allocated memory blocks are padded with identifiable characters which the memory manager can check to verify that no memory over runs have taken place. Calls to the memory block checker can be made at any time in the code so that individual statements causing the error can be identified and corrected quickly.

15.3 Extensive End of Run Diagnostics

The memory manager can be used to produce extensive logs at the end of a session detailing memory usage and errors. This code is not only an error checker but also a runtime optimiser that can be used to streamline memory usage in the game, hence improving performance of the engine as a whole.

15.4 Automated Memory Release Routine

Errors frequently occur as a result of mismatched `alloc` and free `pairs`. Memory can be released on a block basis instead, removing the possibility of a mismatch and also simplifying the end of sub-game and end of level code. Simple code is easier to debug, maintain, and runs more efficiently.

16 Languages

Lego Island 3 will feature a large amount of text, particularly dialogue. This text will be stored in spreadsheet form as this makes it very easy to transfer the database to translation houses. These spreadsheets will then be exported and compiled into a form usable by the main game. Any differences in the way the three platforms use the text will be handled by the export process.

All text in the game will be accessed via a central library to allow languages to be switched simply by changing library.

17 Editor

17.1 Overview

The design for the editor is based on the original Dogs of War dual monitor system but using a much simplified and more consistent approach. It does not include polygon editing tools, texturing etc.

The main aim of the editor is to provide an easy to use environment where the various areas of the game can be changed very quickly without requiring specific programmers to be available to

modify code. To this end, the game engine itself will be integrated with the editing tools to provide rapid feedback for gameplay balancing.

Using this approach it is possible to obtain a very fast edit-test cycle without needing the intervention of programmers. With thirty or more sub-games plus the much larger island environments, tuning the gameplay in response to focus testing and publisher requests will be a very large task and needs to be made as easy as possible. Being able to stop the game, adjust a parameter and then restart immediately will be invaluable.

The sub-game editor is based around the behaviour system currently used on Lego Island 2 with the addition of small number of macros to allow the editor access to the behaviour data structures.

Appropriate AppWizards for DevStudio will also be made available to assist in the development of sub-games and sub-game editor plug-ins.

Most of the new editor will be designed and developed from scratch, however code from the Dogs of War editor LevDev will be used if it is suitable.

Use of the editor is optional, programmers can develop sub-games in the same way as they do on Lego Island 2 but for those who do wish to use the editor it will be distributed using a standard installation package. This ensures that everyone using the editor can get a fully working system quickly.

17.2 System Organisation

The editor will either run on a single card with the game in a window or as a dual monitor system with the editing tools displayed on one monitor while the actual game engine runs on a second. Support will be provided for dual card solutions and dual head cards such as the G400.

Relevant editing tools will be provided as a series of plug-in dialogs. Each plug-in edits a single area of the game, for example, lighting, cut-scenes or object parameters. These floating dialogs can be resized and repositioned as required and store their last known positions. Dialogs are displayed and hidden using toolbar buttons or menu items to create a custom editing environment.

The current sub-game can be started and stopped from within the editor.

The sub-game organisation of Lego Island is ideally suited to this approach, the editor will operate in a similar way to the actual game and switch between sub-games just as the in-game engine does.

17.3 Plugins

Plug-ins will be implemented as statically linked DLLs and turned on and off using toolbar buttons or menu items. This allows development of the plug-ins to be split among team members.

A base class implementation will ensure that all plug-in dialogs maintain their last known position but the plug-ins themselves will be responsible for maintaining any additional state information, the position of splitter bars etc.

When the sub-game editor is run, the plug-ins open when the editor was closed down reappear in the last saved position.

17.3.1 Saving and Loading

Each plug-in is responsible for saving and loading the data it edits. A single “Save” menu item on the main editor window causes each plug-in to save its data.

If a sub-game is closed while unsaved changes still exist the user is asked if they wish to save all changes. If they answer no, they will be prompted to save by each plug-in in turn. A “No to All” option allows all changes to be discarded.

17.4 The Explorer Plugin

The Explorer plug-in is integrated into the main sub-game editor code and is used as a drag and drop source for parameters for plug-ins.

The Explorer consists of two parts, a directory tree and a file list.

17.4.1 Directory Tree

The directory tree is hardwired to the data structure for the game and contains data directories for all the sub-games in the project.

Selecting a sub-game in the directory tree allows the user to close the current sub-game and load the new one for editing.

17.4.2 File List

The file list shows all the files in the current selected directory. These files can be dragged into another plug-in as required. For example, meshes shown in the Explorer might be dragged into an object parameter plug-in.

Only files that will be used by several areas of the game will be listed. For example, lighting files would only be edited by the lighting plug-in and would not be required by any other plug-in so would not be listed in Explorer.

17.5 Object Parameter Editor

The second key plug-in will be the Object Parameter Editor.

From here any game objects in a sub-game can be modified. The exact parameters available will vary depending on the object and sub-game and the programmer responsible for the sub-game is also responsible for making the appropriate variables available to the editor.

Support for the editor is transparent. All the programmer must do to support the Parameter Editor is declare the behaviour data structures using some predefined macros rather than a standard C structure. The macros define a standard structure that can be used by the rest of the game as normal and an editor specific data structure used to provide the edit functionality.

17.6 Example Plugins

A variety of other plug-ins could be created for the system, including

- Object positioning and parameters
- Behaviour parameters
- Waypoints
- Environmental effects, sky, water
- Lighting

- Cut scene editor
- Particle Effects
- Sound parameters

Plug-ins are not limited to tools that edit game data, the editor is a good place to bring together all the tools associated with the game and make them available in a consistent manner. For example, WAD builders.

If a tool that already exists does what is wanted, there is no need to rewrite it as a plug-in although if source code is available for the tool it shouldn't be too arduous to do so.

17.7 Data Formats

The majority of game data will be stored in a text based format similar to that used by Windows .INI files:

```
[Section 1]
Key1=value
Key2=value
…

[Section 2]
AnotherKey1=value1,value2,value3
AnotherKey2=value
…
```

Each data file may have any number of sections and each section may contain any number of keys. The order of sections and keys is irrelevant although each section should appear only once.

Section and Key names are made up of letters, numbers or spaces and are not case sensitive.

Values of keys may be text, numerical or Boolean (Yes/No). Numerical values can also be specified in hexadecimal or binary using the standard C syntax. If multiple values need to be associated with a particular key they can be separated with a comma. Strings containing commas should be surrounded by double quotes.

Default values that are used if a particular key is missing are defined within the code.

Using a text based format makes the data files human readable and allows them to be edited by hand if required. It also allows comprehensive and meaningful diagnostics information to be created automatically.

Game data will be grouped together and stored in larger files for faster loading times.