An Extensible Framework for Designer Driven Procedural Content within Games

by

Andrew Ritchie, BSc

Dissertation

Presented to the University of Dublin, Trinity College

in fulfillment of the requirements for the Degree of

Master of Science

University of Dublin, Trinity College

September 2009
Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Andrew Ritchie

September 8, 2009
I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Andrew Ritchie

September 8, 2009
Acknowledgments

I would like to thank my supervisor John Dingliana for his help and suggestions with this project. I would also like to thank my friends and peers in the Interactive Entertainment Technology course for a truly unique experience. Most of all I would like to say thank you to my family for their unconditional support.

Andrew Ritchie

University of Dublin, Trinity College
September 2009
Modern game development can often be a very long and complex process. It is not uncommon for developers to be faced with creating vast amounts of content to be used in expansive environments that must all be presented to the player through meaningful interactions and often detailed story lines. To aid them in creating and designing such demanding games a developer often uses tools that offer them powerful ways to interact with their content or simplify the process of creating that content.

With the amount of processing power currently available to game developers there has been an increase in the use of procedural content and tools that harnessed procedural features. By using procedural content creation developers are able to use a series of rules and algorithms to represent their assets and then leave the CPU handle the actual generation of the content. Currently with most tools this content is often generated inside the tool in a native form then exported to a format that can be used
by the developers editing tools.

One of the weaknesses with this is that the content must still be exported to a storage format of some type. Another weakness of most procedural content tools is that they are limited in scope and often only implemented self contained and stand alone applications. What this project proposes to do is create a framework to break the limitations of scope and usage by defining a generic structure for creating flexible and scriptable procedures that can be integrated directly into a developers tool chain and game technology.
# Contents

Acknowledgments iv

Abstract v

List of Tables x

List of Figures xi

Chapter 1 Introduction 1
  1.1 Goal .................................................. 2
  1.2 Motivation ......................................... 2
  1.3 Document Outline ................................. 3

Chapter 2 Background 4
  2.1 Procedural Content .............................. 4
    2.1.1 Geometry ..................................... 6
    2.1.2 Images ....................................... 8
    2.1.3 Animation ................................... 9
    2.1.4 Further Uses ................................ 10
    2.1.5 Summary ..................................... 11
  2.2 Flow Based Design ............................. 12
    2.2.1 Theory ....................................... 12
    2.2.2 Visualizing Flow Based Design ............... 12
  2.3 Game Scripting ................................. 16
Chapter 3 State of the Art

3.1 Game Development ................................................. 18
  3.1.1 Procedural Content ........................................ 19

3.2 Tools Development .............................................. 21
  3.2.1 Procedural Asset Generation ............................... 22
  3.2.2 Scripting and Logic Control ............................... 24

Chapter 4 Design

4.1 Procedure Definition ............................................. 25
  4.1.1 Initial Design .............................................. 25
  4.1.2 Revised Design ............................................. 26

4.2 Framework Design ................................................ 27
  4.2.1 Extensibility ................................................. 28

4.3 Implementation Requirements .................................. 30
  4.3.1 Runtime System ............................................. 31
  4.3.2 Editor System .............................................. 31

4.4 Technical .......................................................... 31

Chapter 5 Implementation

5.1 Interfaces .......................................................... 34
  5.1.1 Module Interfaces ........................................... 34
  5.1.2 Runtime System Interfaces ................................ 37
  5.1.3 Editor System Interfaces ................................... 39
  5.1.4 Application .................................................. 41

5.2 Procedure Implementation ........................................ 42
  5.2.1 Structure ...................................................... 42
  5.2.2 Storage ........................................................ 45

5.3 System Implementation ............................................ 47
  5.3.1 Outline ........................................................ 48
  5.3.2 Components .................................................. 48

Chapter 6 Evaluation

6.1 Tools ................................................................. 51
  6.1.1 Managed Code Wrapper ..................................... 52
List of Tables

5.1 ProcedureFileFormat ........................................... 46

6.1 Listing of Editor Callback Events .............................. 55

C.1 ProcLibManaged Native Wrapper .............................. 82
C.2 ProcLibManaged Procedure Wrapper ........................ 82
C.3 ProcLibManaged Implemented Procedure Wrapper ........... 83
# List of Figures

2.1 Two Dimensional Perlin Noise ................................................. 6  
2.2 Grammar Based Procedural City[20] .............................................. 8  
2.3 Simple Example of a Flow Based Design Diagram[26] .......................... 13  
2.5 IBM’s OpenDX Interface for Visual Programming[16] .......................... 15  
2.6 Evolution Robotics Behavior Composer[33] ......................................... 16  
4.1 Initial Procedure Definition .......................................................... 26  
4.2 Components of a Procedure .......................................................... 27  
4.3 Usage Diagram of Library ............................................................. 30  
6.1 Module Load Request ................................................................. 54  
6.2 ProcLibManaged: Procedure Editor .................................................. 56  
6.3 Sine Wave Diagram ................................................................. 59  
6.4 Sine Wave Results ................................................................. 60  
6.5 Image Loader Diagram ............................................................. 60  
6.6 Image Loader Results ............................................................. 61  
6.7 Mesh Loader Diagram ............................................................. 62  
6.8 Mesh Loader Results ............................................................. 63  
6.9 Terrain Importer Diagram ......................................................... 64  
6.10 Terrain Importer Results .......................................................... 64  
6.11 Terrain Generator Diagram ....................................................... 65  
6.12 Terrain Generation Results ...................................................... 66  
6.13 Building Generation Diagram .................................................... 67  
6.14 Building Generation Results ...................................................... 67
Chapter 1

Introduction

Procedural content is a topic that has featured extensive research and usage within the games industry. It has found uses in many areas within the development process as well as within games themselves at run time.

Using procedural content allows developers to explore beyond the boundaries imposed by static assets and systems. Procedural systems have provided developers with the ability to create complex materials, expansive landscapes and even infinitely large universes by simply using algorithms to define features, shapes and behaviors.

Procedural methods however are not limited to use within games themselves. There is a lot of research and development put into dedicated tools for aiding developers in harnessing complex procedural algorithms required to generate content and assets. These tools often focus on providing designers some form of control to help with creating and exporting content that would otherwise have had to be made by hand. Some of these tools and research include foliage generation[21][37], atmospheric effects[10] and even complex city generation[27][13][29].

Until recently it was common that developer controlled tools were entirely separate from the development tools of the game itself. Quite often the game tools and editors would simply be used to import and place content rather than create it. The separation of these tools required that the procedures used by tools to generate their content was limited in scope to just that tool. Moreover the content itself would have to be exported to a format that could be imported into the game’s development pipeline for use. There is a trend in game editing tools to include more controls and creation features within
them, there are even currently some game developers who include forms of procedure definition and procedural content directly within their tools. Though the scope of these features is often limited to a specific system or element of the game or the technology it is working with.

1.1 Goal

This project and dissertation focuses on the development of a generic framework that can be used by developers to expose any of their games technical systems for use in designer controlled procedures.

The goal of the project is broken down into two objectives. The first objective is to design a formal definition for scripts that define the architecture and operational flow of procedures that can be used by game designers to define abstract representations of the procedures. This definition features heavily the Data Flow design paradigm as it is a feature of many existing tools that developers use.

The second objective is to implement a framework that manages the defined interfaces to allow developers to load and run these created procedures directly from within their own engines and tools. Modularity is a key foundation of the framework in order to enabled developers to greatly extend the available features that can be used by the procedures. The framework should also allow developers to create their own collections or groupings of processes that can be organized in accordance to how a designer would expect them grouped.

1.2 Motivation

Procedural content has become a valuable part of the current generation of game development pipelines, though it largely remains a tool used to create specific features or assets. Many content stand alone procedural tools focus on specific types of assets which are then incorporated into games through traditional importing systems. By integration procedure creation and operation into the very core of a game engine designers are able to access and leverage the features of procedural content while still working within the confines of their tool set and engine’s features.
By creating a system that has well defined interfaces that can be integrated directly into developer’s technology this project aims to extend the designers overall control and usage of procedural content as well as explore new uses for it.

Procedural content in games is currently receiving a lot of research from many different game developers[5] and this project explores a way of facilitating possible shifts by developers to using it more extensively by offering a modular system that can be integrated into their technology so their exploration and development of procedural content can be focused on the content itself rather than trying to implement systems to handle the procedures. It is believed that if a system is proposed and incorporated as such a key component to their technology and tools then game developers will be both more willing and better equipped to explore proceduralising game elements, systems and features than they would otherwise have.

1.3 Document Outline

Chapter 2 provides a brief background of some of the topics that provide motivation for the project. Chapter 3 is a review of some of the state of the art applications and uses of procedural elements and visual scripting. Chapter 4 discusses the design concepts and structuring behind the library and associated systems that are implemented. Chapter 5 covers the implementation components that were required to create the ProcLib library. Chapter 6 contains an evaluation of prototype applications developed to use the library and its framework. The final chapter includes both conclusions about the project as well as potential future work.
Chapter 2

Background

Throughout this project there is a focus on defining a structured bridging point for concepts and features that have become important to the growth and improvement in the process of developing games. Each of the topics mentioned has its own interesting and varied history, implementations, uses and benefits for developers however to provide a full and in depth background of each would simply beyond the scope of this write up. Instead, the background for each of the topics is given with a view to specific uses that they have provided to games and the developers.

2.1 Procedural Content

Procedural Content is an area in computing that is both fundamental and extensive. Creating procedures is one of the most basic functionalities of most programming languages, with the entire Procedural Programming paradigm built around it. However procedural content is much more than that, it is a way of abstracting all the detail that an object or even a full scene could contain into an algorithmic form which can be invoked to rebuild the complex data[8]. Controlled and predictable variation can also be introduced by simply using parametric controls within the algorithm.

In the early days of computer graphics procedural techniques were used to provide improved detail and complexity in computer rendered models that only had simple structures or features initially defined[34]. The abstraction of surfaces into algorithms not only meant that data could be mathematically modeled rather than explicitly
shaped but also alleviated the need to store such complex models on disk. Early hard drives and storage formats were much smaller than currently available so making optimal use of available space was crucial. With procedural content the abstracted surface definitions could also be reused thus if large or commonly used but slightly varied objects can be algorithmically represented they no longer need to be stored and can be recreated with variable parameters.

This simple algorithmic representation of content was very heavily used with regards to creating two dimensional textures and surface materials. With relatively simple algorithms realistic textures could be created as needed and in some cases even provide as realistic results as any artist could create. Examples of good procedural textures are features like marble, wood, bricks, clouds and even water due to their geometric and reoccurring features.

Other graphical advances that featured procedural definitions can be found in the likes of particle systems. The definition of a particles behavior and algorithm to manage a particles life span can result in complex and visually appealing formations of spatial effects. The control of a particle system are all based on the algorithms that define them and the initial parameters which is one of the most fundamental components that build up procedural animation systems.

While not limited to just animation and texturing procedural systems are often heavily reliant on variations in parameters, also known as noise, to create the augmentations and differences in results between each generation process. A good noise function can result in drastic changes with procedures, however to retain control and creation of predictable results noise functions also tend to retain pseudo randomness. Not being able to reconstruct a result due to a lack of control can devalue to the appeal of using procedural generation.

One of the most widely known noise algorithms is Ken Perlin’s self titled Perlin noise function[31], presented in figure 2.1 as a two dimensional image. There are many variations of noise algorithms, combinations of noise algorithms (such as fractional Brownian motion[9]) and even inclusion of variance into more complex systems like fractals.
2.1.1 Geometry

Terrain

In the graphics and games industry one of the most common uses for both noise and fractal variations of procedural content is in terrain creation. Given the complex and vast nature of terrain procedural methods are employed to save both time required to manually create them as well as save on storage space that would be required for vast environments.

Some systems use procedures to generate texture based terrains in the form of height maps or fields. A height map usually refers to a 2D gray scale image where as a height field is simply a two dimensional array, which are scalar or gray scale image representations of the terrain usually with low values being the lower points and solid white being the highest most point. From this image each pixel is used to define the height of a vertex or part of a mesh.

Using procedural content combined with height maps is useful for developers as they can generate the height maps with a procedural function create their mesh from that. They can even employ multiple resolutions of height maps to create variations in the detail levels and use them for explicit level of detail systems such as quad trees or other precalculated polygon reduction method[19].

Continuous terrains are procedurally rendered. This means that rather than pre-
defining all the vertices and faces an algorithm will calculate the intersection point of the terrain using the generation function. Commonly this is done using a ray marching algorithm which is similar to ray tracing[12] except instead of examining an existing scene for ray collisions it uses a step based iteration to move the ray forward and with each step processes a procedure to calculate the terrain height at its given location.

**Foliage**

Geometry generation was not limited to large scale recreations of natural features; it has also been used to create living objects such as trees and plants to populate those environments. Foliage generation harnesses a procedural system that is built around generation or definition of a grammar known as a Lindenmayer System or L-Systems[32]. Grammars in procedural usages tend to be made up of constant symbols that indicate an action to perform or procedure that manipulates geometry directly, variables that can be used to control recursion or as parameters to operations and production rules that tend to be assignments to associate the variables with a series of variables and operations.

A simple example would be. Define ’+’ to mean rotate 10 degrees right, ’-’ to mean rotate 10 degrees to the left and ’—’ to mean move up two units. These are the functional elements; they incur a manipulation of the geometry directly. By defining control sequences of A to contain ’-B+B+B’ and then define B to contain ’——A’ this can be used to iterate to a provided depth all the while generating a structure that spawns three branches which all extend upwards two units at different angles and spawn three more branches. This system combined with controlled recursion incremental step based generation of data that self manages the next step.

This method of procedural generation has proven very useful for plants and natural formations that grow and branch. Trees and plants can be represented as simple recursive branching from a main trunk with the lowest most branches containing or being tipped with leaves. This system also provides the ability for developers to add levels of details by either simplifying the steps or using an entirely different set of production rules depending on the viewing distance.
Buildings

By extending the grammars used in L-Systems developers have been able to successfully generate geometrical representations of manmade structures such as buildings and their facades[23]. By combining the building geometry generation systems with a distribution and placement procedure it has been show that entire cities can be created as seen in figure 2.2.

The geometric structures of procedural buildings are rarely complex or include interiors due to the costs required to generate them and store them, especially when the environment contains many hundreds of structures and other resources. There is currently research and development being done into exploring the possibilities of using the algorithmic properties of procedural content to create vastly more complex structures through the use of lazy loading. Lazy loading is a technique that loads or generates assets only as they are needed rather than creating them at load time and caching them. Procedural systems like these are becoming more popular as the power and available processors on CPUs and GPUs increases.

2.1.2 Images

Procedural methods have been put to many uses with image generation[30] ranging from simply generating repetitive patterns to generating complex and unique works of
Procedural imaging is often employed with terrain generation with regards to both the creation of height maps for creating the terrain and the textures that are placed on the terrain surface. Procedurally generating surface textures is often done by using look up systems that have a series of pre created textures that are used to determine colour values at specific pixels based on the height map value at the corresponding pixel and some form of blending function[15][2].

Image generation has been used beyond even the simple look up tables for generating complex visuals, natural features like grass and fields have been procedurally generated[7]. Methods for generating realistic skin that can be used to texture polygonal models[17] have been developed even as early as the mid nineties, preceding even the artists quality game creations. Procedural patterns can be employed to generate facades for buildings[36] as well as natural surfaces like rocks and granite[38].

Developers have used algorithms to procedurally generate and alter images in non photo realistic appearances[25]. By employing algorithms to recreate the look of paintings or images that abstract artists would create developers are able to create a visual theme that could be unique to their project yet flexible enough that artists needed spend time changing their work flow to create such a style.

2.1.3 Animation

Object animation is an area that procedural features have been able to aid greatly. Animated items like particle systems, as previously mentioned, are brought to life through procedural simulation. Even pure physics driven animation can be thought of as a form of procedural animation, simply a special case where the procedure is built up of the real physical rules that objects adhere to. However developers have used procedural animation bring more ‘life’ to environments and even characters within those environments. By defining rules for object animations, like trees swaying in the wind, developers are able to create realistic foliage that reacts to variations in wind without having manually create animations for all possible strengths[14]. Using procedural response systems developers are able to combine and manipulate facial features of characters in was that can convey emotion and even sentience to the viewer in real time and dynamically based on events as they happen[6].
2.1.4 Further Uses

Procedural content is not just limited in use to graphical or rendering related features either. There has been development in many areas that explore the ability to use procedural methods to build and structure data that can bring complex interactions and behaviors into an application or game.

An example is in the area of storytelling where researchers have implemented systems to procedurally produce stories that allow the users input to not only manipulate specific situations but also effect the entire story arch[24]. By using procedural methods to built stories or branch events within a story developers are able to offer players decisions and control of events where previously they would have been offered no or very limited input.

Artificial intelligence is another area where procedural methodologies have been able to bring interesting features to light especially for games. Though AI in and of itself is simply a massive field of study that can offer many possible examples of implementation of procedural methods the most obvious is the processing component of intelligent agents themselves. Agents, as defined by Russel and Norvig in their De facto textbook on AI, use percepts from an environment which results in actions performed by that agent[35]. These agents are further classified based on how they use these percepts to determine their actions, but the foundational component is that for developers implementing an agent their logic is a procedure. So by using this agent methodology, using percepts as the data for the procedures and the actions as the results, for implementing game AI beyond simple explicitly scripting or event triggered entities developers were able to explore creating AI for their games that could behave based on adaptive procedures that they created. The AI within games is a prime example of games originally making trade offs for processing power that needed to be used for graphics for complexity of other systems. However as processing power became more available to these systems developers were able to use research from pure AI fields and use the procedural systems to give their non player characters far more complex behaviors.

Sound systems within games have also benefited from procedural features. Similar to features like geometry data, music originally started on computers as entirely procedural as the only way to include music in an application that could run in the small
memory space was to use an algorithm to call the appropriate notes or sounds. However as playback abilities became better and compression/decompression algorithms improved music was soon externally created and saved for direct playback by the game. This soon became difficult to manage as the time required to create music combined with the amount required by the game would cause developers to reuse music within the game. While it was possible to create short looping tracks that could be used to minimize the space required, it could be difficult to invoke an emotion or match the game state at the time. Developers began using procedural methods again to manage the game music, so instead of using procedures to generate the music itself the game would use events or game states to determine what to play. This type of procedural content would further develop with systems becoming much more complex and broad enough to be able to potentially generate a complex sound track matching the players experience of the game without the designers having to specify events and simply defining the rules that game used to trigger musical cues and tracks.

2.1.5 Summary

Over the last two decades procedural content has flourished and expanded into many different areas of game and graphical content. However with all the research and successful implementations of procedural content there are still always pros and cons of using it within games.

The general strengths of procedural content are usually highlighted by the ability to reconstruct complex data from the relatively small space requirements of an algorithm. Though with content creation there can also be time savings in using an algorithm to generate data that would have otherwise had to be manually created. The reusability of procedural algorithms is truly beneficial when a process has to be repeated many times, especially when the process repetitions require small variations at certain points which can result in drastically different results. The parametric control of a procedure can be of great benefit to the implementation and usage of procedures.

The weaknesses or trade offs for procedural generation however can be quite important for developers to take into consideration too. Procedures tend to be much more processing intensive when generating when compared to loading or streaming the data from an existing source. Procedural content is often associated with higher memory
requirements, especially when the data that it is to generate needs to be stored for use after generation. Most of all though procedural content is entirely dependent on the abilities of the developers. A procedure cannot do any more or less than what it is created to do, even with extensive or complex parameters the algorithm that is developed must be able to meet its needs otherwise the data it generates could potentially be of no use and typically with procedural content this puts a lot of burden on the programmers to be able to accurately assess the application designers needs.

2.2 Flow Based Design

2.2.1 Theory

The core structure of the procedure definition is a variation on Data Flow Design[1] or simply Flow Based Design(FBD) or sometimes Flow Based Programming(FBP) depending on the context. The core concept of FBP is the messaging or data transmission occurring between isolated processes via ports which are the only exposed features of a process. The processes themselves require no formal description so are treated as ‘black boxes’. The ports that can be exposed by processes are defined as either input or output. An input port is a recipient for information being sent to it and the output ports are transmission points, generating data to be sent to input ports, usually on other ports. Figure 2.3 shows an example of visualizing a simple flow based diagram. Analysis and design of a FBP system is very straight forward due to the simplicity and directed nature, yet each process within the system can be very complex as can the data being transferred between processes without effecting the structuring or simplicity of the system.

2.2.2 Visualizing Flow Based Design

Flow based design lends itself very well to visualization. As the complexities are abstracted as well as the structures and concepts lending themselves very simply to be transposed to a visual formatting as was mentioned in the previous section and seen in diagram 2.3. Extending upon this developers are able to abstract the implementation behind these visuals and present it to designers or non programmers to manipulate in
order to create the appropriate paths between processes as they see fit, never needing to worry about what is truly occurring in the background. This visual formatting and interaction has often been described in terms of nodes [4] instead of processes.

It should be noted however that this project does not propose any specific interface structure but the research and design of the structure is heavily based on the following visualization topics. The visual use of data flow in the application design process has been in research and development for many years [18]. So by heavily relying on the data flow design process the project also takes into account the following exploration, development and successes of the adaption of visualized flow based design.

**Visual Programming**

Visual Programming is an expansion on the concept and usage of visualizing data flow. Flow based design in and of itself does not define an specific form of control sequence or defined branching, it is purely ignorant of the goals of the system or purpose of a node. Visual Programming however implements a programming like set of nodes to act both logically and practically like the text based programming language equivalent syntax. Figure 2.4 shows an example of a visual programming language ‘CODE’ designed to allow visual creation of parallel applications. Along with the control nodes visual programming systems usually bring in more strict definitions or identification for the type of information that is passed between nodes. This control provides the ability to generalize nodes and their internal operations to a finite set of possible inputs. The strengths of a visual programming system owe a lot to the FBP that it is based on.
though, as the ease with which it can be used is due to the abstraction of the complexity that is contained within each process. The user is able to generate high level systems simply by defining the connections and their order between the low level systems. By providing the user with a modular programming system on top of the simple to use concept visual programming is in some cases ideal for Rapid Application Development and Rapid Prototyping.

Visual Scripting

Visual scripting is a occasionally used with reference to visual programming however in the scope of this project it is treated as a separate concept from pure programming, specifically because it has some distinctive and key differences. The primary distinction is that a visual scripting system does not generate executable output or output compiled executables. The scripted systems rely on some other framework or system to give its actions meaning full associations as well as manage its loading and invocation. The most common way to run or develop for a visual scripting system is through a development environment that also includes a runtime component that can run the scripts such as IBM’s OpenDX as seen in figure 2.5. Another common key feature of Visual Scripting is that it is used to control data flow through the presentation of only high level objects. While some scripting systems do incorporate low level program-
Figure 2.5: IBM’s OpenDX Interface for Visual Programming[16]

Programming like features most Visual scripting systems are not concerned with composition of objects or modifying their behaviors but rather about coordinating data passed to and from the objects and using that coordination to dictate the actions or results of the system it is controlling. This usually entails treating complex data types as they would simple types so the user does not have extra complexity as well as avoiding the need to structure features in such a way that it represents an explicit algorithm or program flow chart.

These control systems are only meaningful full within the scope of the application running them as well as not being intended to generate stand alone applications. Through the higher level of abstraction visual scripting is geared far more towards Rapid Application Development[22] or more specifically Rapid Prototyping. These scripting systems became very popular with designers due to the minimal program design requirements and encapsulation of entire systems into single, strongly defined objects. Examples of these uses were in robotics where any single node could represent many complex mechanical operations and data manipulation, yet the designer simply had to interact with it as a behavior associated with the robot as seen in figure ??.

These higher level abstractions of objects allowed designers to create complex con-
control systems without being hindered by the need complexity of combing general purpose data into the complex structures they needed. Loss of generality associated with a programming language feature set allowed for wider accessibility by system designers.

2.3 Game Scripting

Scripting is another area in which games developers have taken great interest. Scripting languages are most commonly intermediary languages that can appear similar to a programming language but are parsed or compiled at run time unlike most fully fledged programming languages that are compiled into a binary format. The advantage of scripting languages in games is that they allow the developers to externalize game or even asset specific functionality. Scripting allows the developers to retain generality within their game engines and allows for flexibility in game specific behaviors. There are several variations on how scripting languages are used within games, on type uses specific keywords and structures so that none of the normal general complexities of programming languages is visible as well as keeping the grammar concise to their game. Often, however, scripting languages can be quite comprehensive, in some cases developers incorporate external scripting languages into their game engines in order to trade of the cost of developing their own and having to maintain. These languages allow the developers the complexity that can allow for scripts that operate above and
beyond the original intentions of the engine without the overhead of including it into
the core engine or tying it to any specific project. Due to its nature scripting is also a
very powerful tool when it comes to development time, designers and scripters are able
to get quick, in some cases instant feedback on script changes without the complex
build processes often associated with embedded and monolithic game engines.

Scripting in games is still an area that is expanding and growing but it has success-
fully proven how successful and useful it can be to harnessing or adapt technologies or
ideas that allow for non programmer developers and designers to manage and direct
game specific features. While not being a pure transition from program controlled to
script controlled games the process of exposing more and more core features to de-
velopers and designers has been occurring over the years as well as development into
the interfaces and interactions that these developers use to manage and control the
exposed scripting systems.
Chapter 3

State of the Art

3.1 Game Development

The process of developing a modern game can often be a long, time consuming and complex process. The average current generation game is made up of many different types of assets which define the shape, look, sound, feel or even flow of the game. Throughout the development process of a game it is quite often required that many different types of content developers create assets for it, the process and order in which the content is created or dependencies upon other content development is referred to as the development "pipeline". These developers most often include the likes of 2D artists, 3D modelers, level designers, game designers, story writers, programmers, sound designers and many more. All of the associated artists and designers are usually familiar with the results of their specific assets, such as a 2D artists would be focused on creating desired images and level designers defining the world in which the game is played, however the data format that the game engine uses to store assets is usually very different from something that the developers are used to working with. For this reason most engines usually have exporters that convert assets from the format that a designer defined them in to the appropriate format that the engine uses. Within the engine these assets are loaded and then converted so it can displayed back to the user in a form more closely resembling what the original designer intended.

With the success of games development lots of time has been spent developing not just more complex and detailed ways of generating game assets but also into the very
tools that developers are able to employ to control that generation. A game developer
may have a full team of programmers who have to work with another team of designers
in order to achieve the desired results.

3.1.1 Procedural Content

Though not a new concept, procedural content is an area of development that has
started to receive a lot of praise from within the games industry. Already used in
many areas, especially amongst those in the demo scene, procedural content is a form
of data generation that involves designing an algorithm that reproduces data using
provided parameters. Often heavily associated with noise algorithms, such as Perlin,
many game assets can be represented as procedural elements. Some examples of game
assets that can be represented procedurally are terrain and textures. The appeal of
procedural content is that by having algorithms capable of building complex assets
from just parameters the total space required for storing the asset is greatly reduced.
The trade off with this saving is the processing power required to run the generation
functions and the memory required both during the operation and for storing the
results.

Procedural content has seen usage in many forms throughout the history of games
development, ranging from simple definitions of line shapes to complex generation of
full planetary bodies. However at its core procedural content has changed very little.
The real changes have come with these varied uses and implementations that developers
have put it to as well as tools and interfaces for creating and controlling the results.
Procedural content has strengths in its simplicity, as a procedure that can change with
every run or alter drastically simply based on altered ’seed’ or parameter values a fairly
concise function can result in a complex data structure.

Developers have used procedural content in games in order to achieve specific goals.
The major advantage of using procedural content in the context of a game is that once
the rules or procedure itself is created the game can reuse it as much as it needs.
Whether it be simply to generate the same terrain every time the game loads or to
generate entirely new scenarios every time the user makes a decisions, it’s as simple as
invoking the procedure with appropriate changes or values for the seeding parameters.
In some cases the generation might be of entire solar systems so that is not to say the
changes are instantaneous but rather the developer can knowingly invoke the procedure and retrieve a predictable output type.

**Infinity: Quest for Earth**

Infinity is an independently developed Massively Multi player Online Game (MMOG) that brings some of the features from games like Elite into the current generation. Infinity is a oriented around space travel and space craft so in order to meet the immersion of truly being able to travel through space the developers have implemented a large number of procedural systems for the game. Planetary bodies are procedurally built, both terrain and texturing, in order to allow the player to travel seamlessly between space and planet surfaces. Entire solar systems are generated procedurally to allow the user to explore the vastness of space, unbounded by the edges of what the developers could possibly design manually. Even the story systems and missions for players feature procedural generation to avoid the need for the developers to manually create story lines for environments they may not even be sure will exist in a certain instance of the game. The procedural content system is as much a part of the game as the rendering system.

**Left 4 Dead**

Left 4 Dead by Valve software is an implementation of procedural content that can be misleading. The AI Director that controls the games zombie hordes is a procedure, it uses collected stimuli and states from the environment and executes events that it sees fit to execute. The desired result for Valve is that their game offers a replay ability that provides the gamer with a different experience, whether subtly or blatantly, each time the play through. The procedurality of it

**Star Wars Galaxies**

In Star Wars Galaxies the foundations of each planets terrain is built around fractal bases. On top of the base terrain developers can create their own more controlled layer. By using procedural content the developers were able to create large scale terrains for the planets that the game presents to the players while using more manageable modifications to define and shape the key locations.
.kkrieger

From the demo team .theprodukkt, .kkrieger is a fully procedurally generated FPS game that exists in an executable only 96KB in size. Relying heavily on procedural algorithms for geometry creation, texture creation, object placement and many other features it represents an attempt it fully proceduralising a game world outside of the space genre. The developers adapted their techniques, technology and knowledge from previously making demos and applied it to the process of creating an interactive game.

3.2 Tools Development

As game developers have expanded their content, asset and scripting requirements tool development has exploded becoming a large and in some cases required part of the development process for artists and developers tasked with creating the content. In order to facilitate this increase in productivity required to meet the expanded workloads developers have devoted time to creating tools and tool chains that simplify the underlying complexity associated with various types of content. Tools are used to provide simpler interactions between the developer and the item they are creating. By simplifying the process exposed to the artists and designers they are able to focus on the asset itself and abstract to some degree the complexities associated with getting it into their game. These tools have evolved in both features and capabilities.

Some tools development is dedicated to creating stand alone tools that are very powerful but focused on a very specific asset type or task which can then be exported to and imported by their game engines pipeline.

Game developers have also developed their own tool sets, some of which are designed to greatly ease the burden of managing all the different aspects associated with a game and the work flow required. Comprehensive tools such as UnrealEd for Epic’s Unreal Engine and Sandbox for Crytek’s CryEngine are examples of editors that incorporate many aspects of the game pipeline and even provide ‘in game’ presentations of the developers work, referred to as What You See is What You Get (WYSIWYG). Other tools such as Hammer for Valves Source Engine are an example of a tool developed for specific development step in the overall pipeline but focused entirely and specifically on the requirements of the game engine. Even as the ”in house” tools become more
powerful, however, they must still act as the bridging units to core engine technology so the burden is upon tools developers to ensure that all features in the applications will result in appropriate outputs in the games, even more of a burden is the need to provide import and possibly export formats to support the powerful but generic external editors.

For this project the primary focus is on integrating procedural content and processes into a developer’s pipeline so the tools that were most closely examined either managed the creation of procedural content or provided a way of presenting the developer with a way managing the design of content using a visual flow design system. While these systems do fail to achieve the proposed integration of procedures that the project outlines they do provide solid and practical examples that it is based on concepts and methods that developers are already familiar with.

3.2.1 Procedural Asset Generation

There are many tools used to create and export assets that have been generated procedurally. A full list of these tools would be both extensive and quickly out dated. The following selection of tools are simply highlights that feature examples of common features that developers are familiar with but also expose the fact that so many tools are purpose built. These all serve to aid the development process using procedural methods of some form by simplifying a complex process but they are also tailored to create specific results.

**Terragen**

Terrain generation is one of the most common uses of procedural generation and Terragen is a standalone tool dedicated to just that. Using Terragen designers are able to provide parameters and even in the latest release use a node based editor to control the formation of the terrain, texturing features and even shader details.

Terragen is an example of a tool that is not limited for use by game development studios but also by any media that requires high quality landscape and terrain rendering or generation. While offering the ability to export height maps, meshes and textures is not it’s primary feature, in fact it’s rendering capabilities are a primary source of it’s development focus. As such it’s most common use by game developers is not it’s
terrain export but the ability to render images that can be used as sky boxes. The immense procedural terrains that the designers can create are rendered to textures or images that the game can import and draw around the player to give the impression that world they are within is much larger than it actually is.

**CityEngine**

The CityEngine tool developed by Procedural LLC is a standalone tool built on the Eclipse framework that is developed to procedurally generate entire cities. The tool is built on extensive research into using procedural grammar to create complex buildings and then using further grammar and procedural methods to organize the buildings into a desired city structure. The premise of the tool is that it can allow developers to quickly create vast city or individual building geometry for use within a developer’s application. For games it features exporting capabilities of a generated mesh so that it can be imported into the developer’s pipeline. While it can be imported as a precreated static mesh, such as Crytek’s Cryengine[20], the abilities of the scripting system are restricted to the development tool.

**SpeedTree**

SpeedTree is a middleware solution that provides the generation and management of high performance and expansive foliage systems into a game environment. The middleware uses advanced rendering capabilities to procedurally place trees within a scene providing them with not only rendered geometry but also shadowing and even animations. The textures used traditionally managed image files but the levels of detail, placement and animations of the trees is dynamically determined by the application as it runs. Using this middleware can drastically reduce the time taken for developers to add vast forests and populate other outdoor scenes with fast rendering trees.

**Euphoria**

Euphoria is one of the most advanced commercially available libraries for procedure driven animations. A unique system used by the Euphoria library drives animations in physically simulated objects by using procedurally generated movements based on
desired goals within the system. Using procedural systems within the animations allows Euphoria to dynamically simulate entities that interact and react in real time.

### 3.2.2 Scripting and Logic Control

**VVVV**

VVVV is a toolkit designed around a visual programming interface and used for video synthesis. The interface uses abstract representations of visual components and manipulations for those components that can be placed, manipulated and connected by the developer to generate desired graphical sequences.

**Quest3D**

Quest3D is a game and interactive virtual reality development environment that uses a node based graphical editor to control the operations of the engine as well as generate hybrid objects. The Quest3D system uses the node based editor to both implement the control structures for the projects and to define extended behaviors of objects.

**UnrealEd**

UnrealEd is a tool that is developed by Epic Games development with their Unreal Engine. The editor integrates visual scripting in several ways. There is an advanced flow driven visual shader editor that allows designers to create material shaders by simply placing various nodes and connecting the appropriate inputs to outputs. The nodes available within the shader editor represent common components used within shader algorithms.

The developers at Epic also utilized visual representation of their level and entity scripting system called Kismet. The exposed connections and nodes directly manage and direct the actions that occur within levels developed for the Unreal engine.
Chapter 4

Design

4.1 Procedure Definition

In order to develop a framework for defining procedural content there must first be a clear definition of what a procedure is within the system. Defining a procedure ensures that its component pieces can be isolated and appropriately interfaced as well as figuring out what the entry and exit points for invoking a procedure will be.

A hybrid structure a for defining procedures was devised based around data flow programming, visual programming and approaches taken existing development tools. A key factor in using this hybrid structure was the emphasis on streaming information from point to point rather than placing any importance on the internal workings of the processing units. Though not implemented in as a visual tool the goal was to use a syntax and definition that transferred easily between visual editors integrated into game designers tools pipeline and the low level processing library.

Several important iterations were used to fully design all the required components and interactions. By using a minimalistic approach each iteration focused on only incorporating the most broad and abstract components that would be required to provide a flexible, extensive and generic procedure implementation.

4.1.1 Initial Design

The initial design defined only three objects, as seen in figure 4.1 in the form of nodes, sockets and data streams. Data streams are simply wrappers for data that is to be
passed from one location to another. The streams are logically directional in that when they are written into from a single location and that data is sent to the recipients. Sometimes stream is also referred to as a connection due to the logical roll it plays in bridging a senders and receivers however it must be noted that there is only ever one sending objects even though there may be multiple receiving objects. A socket is the object that manages the transmission and receiving of data by a data stream.

The sockets are the interface between a node and the rest of the system. A node, in turn, represents a processing component within the procedure. Nodes encapsulate a core algorithm, data manipulation or any other function that a developer may require. There were also specifications for sockets that represented external system inputs and outputs that managed data flow to and from the procedure.

### 4.1.2 Revised Design

While the original design was based heavily on achieving a pure data flow oriented system it became clear that to be useful in games as well as designer or scripter friendly it needed some extra features. These features were the ability to define and handle external system interfaces as well as providing parameters. These additions can be seen in figure 4.2. This divergence removes some of the generality of the data flow system as the various components have significance but what they accomplish is providing external interfacing to complex systems for retrieving and setting persistent data or simply provide access to information outside of the procedures scope. The data traveling to and from nodes and referenced interfaces can still be referenced as a data stream but
the interface definition allows for more direct and specific operations and data sharing. The parameter feature is a specialized socket capable of taking data from a stream within the procedure or using a default hard coded value. The parameters are similar to traditional properties in most editor systems, apart from the ability to be overridden by data streams, these allow the developer to explicitly define certain parameters as well as avoid cluttering the procedure with data streams or inputs that are always going to be static or constant values.

The revised design was based mostly on the concept that procedures may retain complex data that needs to be conveyed to the game or higher level system but cannot be predictably expected as an output by the procedure invoker. It also features some additions made to integrate similarities from existing systems that could ultimately improve performance and can help tools implementers separate constants or parameters from rest of that data that is being propagated through the system.

### 4.2 Framework Design

As the development goal was to create a framework that could be incorporated into a developer’s core technology there were a few components that were viewed as priority needs for the system. The primary focus of those components was on ensuring that the library could easily be extended to incorporate many different types of features and processes that the developer might require for their game. Without this extensibility developers would have limited flexibility in using the system and ultimately hinder
their ability to pursue new procedural possibilities within their editors and games.

4.2.1 Extensibility

To ensure the desired extensibility there were a few general concepts that were given priority. Mostly they focused on ensuring that the system did not directly require many of the specific data, formatting and implementation details of the systems the developers implemented. There was also importance placed on the ability to allow developers to componentize their implementations so that they could easily reuse elements as well as switch out various collections or individual components depending on what the designer required or the type of content they were working on.

The high level priorities were placed on:

- Generality
- Flexibility
- Modularization

In some cases these concepts are synonymous with good software design but in practical implementations can become lost during the growth of a project. With the previously mentioned design focusing on a minimalist approach these goals would not only be easier to keep track of but also facilitate and help support each other.

Generality

One of the key differences between true Flow Based Programming and Visual Programming is that FBP is not in any way interested as to what the data that is being passed between processes is. In this respect visual scripting systems tend to be similar to visual programming in that they usually have stronger definitions and interest in what the data type of streams is. It is the same with most visual scripting systems, in that there are usually a set amount of strongly defined data types that can be used within the system.

Within ProLib’s framework there is no strong enforcement of definition of data types. Due to the focus on games it does define common formats that might be used however these are not required and the data being passed through the system can be
any that the developers create. This generality can be limited in some regards by providing the ability for the developers to identify specific data types that they wish to ensure, however by default this is not to be required.

Using the weak typing and not specifying any internal structure for data streams leaves the system open to allow for expansion without changing the interface while still retaining some form of control for when it matters to the user and process. This system also enables tiering of data types so that developers can implement abstract interfaces to certain types of data then be able to implement concrete objects from those interfaces. This kind of flexibility is a primary goal of the generality, to provide only control and direction rather than defining implementations of data.

**Flexibility**

The flexibility of the project is very closely tied to the generality. Flexibility in this context is the attempt to minimize the limitations placed upon what developers can do with the system. As the processing nodes in a data flow system are ideally black boxes it would be implicit in the flexibility of each element however when a designer needs to control, direct and manipulate their procedures it can quickly become difficult to manage certain events or access data that would not normally be available to a flow based system or possibly even a true visual programming system. Therefore the inclusion of a small subset of interfaces and manipulatable objects that nodes can access opens up the options available for a designer to use ideally without having to resort to bypassing the procedure structure itself or hacking features into nodes that break their generality.

By focusing on the game development process the inclusion of a small set of slightly more task oriented interaction objects helps retain the generality of the general flow process. This flexibility ideally allows for developers to use procedures that can vary from data generation, to chunks of game logic and possibly even implementing game system control procedures.

**Modularisation**

The extensibility there arises an organizational issue. The design of the system takes this into account by incorporating modular containment of node definitions. This is
done in two different ways, the primary being as external libraries that the runtime component is able to load and retrieve data from and also by providing a registration method that an implementing application can use to explicitly pass a registration interface to the runtime.

By providing the ability to separate the implementations of nodes the library aims to allow developers to reuse these modules as well as organize them in meaningful collections that designers can use rather than having to rely on a monolithic or explicit system that contains all possible node definitions.

### 4.3 Implementation Requirements

When implementing the run time components it was planned as being developed using a single common library that provided only two external interfaces. These interfaces would be used for interacting with the library in an edit mode and a run mode. As seen in figure 4.3 the game developer’s technology is free to access either or both interfaces depending on its needs. However the editor system is a super set of the functionality and features of the normal run time so if the game itself is in an edit mode the ability to run and load procedures identically to the normal run time is still available, it just has more editing options available.
4.3.1 Runtime System

The main runtime system is designed to serve two main uses. The first is to provide an interface for the calling application to request procedures to be loaded. The runtime internally manages the structure, creation and organization of all associated components for these procedures as well as ensuring that required external modules are loaded and available. The other feature it offers is the retrieval of loaded procedures. This allows an application to explicitly request an interface to a generated procedure so that it can invoke the run operation.

4.3.2 Editor System

The editor interface is what a game’s designer tools would use to allow developer interaction and manipulation of procedures. The editor provides the same functionality as the main runtime as well as a much greater extension to the available manipulation options. The editor is able to accept requests to load individual modules outside of the context of any particular procedure; it also accepts requests to create both individual procedures as well as groups of procedures. Within individual procedures the editor also allows for explicitly requests to add and remove nodes as well as create and destroy connections between nodes, which is how the designer would be able to control and manipulate the structure of a procedure during their editing process. On top of the ability to load procedures the editor provides the ability to export created procedures to file as well as update existing ones. With all the possible changes that can happen during the life time of the editor it ensures the application that it is interfacing with is able to receive notices about the changes through a collection of exposed callback routines. The callbacks range from module loading and removal events right down to modifications to connections between nodes. While not a mandatory part of the system the callbacks do ensure that it is possible for the external editor to be aware of any internal state changes that occur.

4.4 Technical

In order to retain a high likelihood of ensuring compatibility with a game developers core technology the project’s core library is implemented in C++. Due to the nature
of the project and the limited time frame it does use some features of the language that developers may not ideally use however these are mostly internal. The core features are implemented using Visual Studio 2005 and can be compiled as Dynamically Linked Libraries (DLL) or statically linked libraries under Windows. The file format for storing procedures is the eXtensible Markup Language (XML) as the formatting is structured as well as being both easy to manipulate and extend.

For external modules they are also implemented in C++ again being compiled as DLLs under Windows. However C# is also used for a portion of the library in order to match the current trend in game developers who are implementing their tool chains in managed languages. The C# component is again developed within the Visual Studio under Windows and implemented as a managed DLL to allow for bridging of the native C++ used within the library and the CLR used in the C# elements.
Chapter 5

Implementation

The original design and plans for the project were radically different than what the resulting ones were. This was heavily due to taking an approach that was structurally very unorthodox. By using a loose proposal and prototyping with short iterations prior to laying out the full technical design the project’s focus and goals were narrowed down to a manageable and practical subset. Procedural content and designer interaction are very large topic areas so by using rapid prototyping that focused on refining the projects design and identifying specific features or areas in which it could have the most benefit. The projects overall design was narrowed down to what would maximize the usability of procedural content for the implementing technology but also minimize the implementation complexities for a developer.

The overall structure of the library and its components are laid out in figure A.2 which can be found in Appendix A. This structuring outlines the 3 major separations of implementation elements. These parts are referred to as:

- Client
- Module
- System

Through their interdependencies the Module and System components, which can be seen in detail in figures A.4 and A.5 respectively, make up the operational components of the library which is referred to as ProcLib. The Client, found in figure A.3, is
the game or application developers own technology which interfaces with the System component and invokes the operations and actions of it. The module is also developed and implemented by the developer for their own specific needs. While all externals interact with the system via interfaces, found in figure A.6, the actual management and creation of components is handled through the implementation which can be seen in figure A.7.

These system components, their individual purposes and the implementation of the controlling management system are all discussed through this chapter.

## 5.1 Interfaces

The implementation is broken into three parts, consisting of the internal library components, the external developer implemented modules and the interfacing or bridging classes that ensure the interoperability. The interfaces to manage the interactions play significant roles in allowing for the goal of extensible and generic procedures to be implemented. They also aid in allowing for dispersal of operation and data types across multiple collections or modules as they are referred to within the system.

The most crucial and common interfaces can be broken into three categories: common, runtime and editor. Within the context of the ProcLib and most of this project runtime is referred to with context to the user runtime or application operation. This means when the application is being used, which is unlike the editor mode which refers to the library being used by the application developers to manipulate or create procedures. The runtime and editor interfaces are often extensions of one or the other, most commonly editor interfaces extend runtime interfaces, in order to reuse as well as ensure that core functionality does not drastically change between modes.

### 5.1.1 Module Interfaces

The structuring of the interfaces for the module component of the system is based around developer extensibility. This component is how the developer implements their extensions and additions to the system, which can then be used by the core system.
IDataStream

The interface to data streams is built heavily around the concept of the data flow design. The interface represents some type of data or array of data that is to be passed between processes or nodes within a particular process.

In order to facilitate easy identification and if needs be control based on the data type the data stream interface defines a simple type identification ability as well as sub definition to allow for variations on common types. An example of this might be an image data type with a further definition of being a jpeg image type. There are also methods that allow for identifying if the structure is a multi element array and how many elements it contains. A definition for accessing a raw pointer is available to allow developers to obtain low level memory access should they need it.

On top of identification and access the data stream interface also defines control methods to help with referencing stream instances, thereby allowing a single stream object to be passed amongst multiple processes. This ensures that the a new stream object is not always required to simply be passed to another node as well as handling deletion automatically and locally when the last reference is removed. The data stream also provides cloning and snatching features. A clone is the process of creating a new data stream that contains a new instance of the data that the original contains, while a snatch results in a new data stream that contains the data of the old stream and resets data reference of the old stream.

This interface is one of those that is implemented by the developer there by giving them control over the management of their custom data.

IStreamRegistration

The stream registration interface is a core bridging interface to allow the main library system to access, identify and instantiate the data stream objects that are available within a module. Upon implementing a new data stream the developer registers the stream class and details with the local module’s registration object. When the module itself is loaded the system requests this module and then uses it to access the data that the developer provided for each of the available data stream classes.
**INodeWrapper**

One of the most important classes for the entire system is the node wrapper. This class is instantiated by the core ProcLib system and associated with one of its internal node components, however the actual operation of the associated node simply calls into the ‘OnRun’ operation of the node wrapper class.

This is another interface that the developer implements themselves but that is how the system is able to retain its generality and extensibility. The interface defines operations for various points of its associated nodes life cycle to ensure that when it is invoked the data it is operating with is valid.

Within its run operation the node wrapper can ensure that it is able to access input and output socket interfaces of its associated node both a system interface that is mentioned later. The input sockets can be read from which returns a data stream interface and the output sockets are able to have data streams written to them or if they contain an existing one this can be modified.

It is also through this interface that the application running the library is able to pass its own information such as interfaces and other specific data. Upon initialization a reference to application defined data is passed to the node and through this reference it is able to integrate more closely with the application. These can be anything that the developers desire ranging from custom instructions to access to resource managers.

While breaking a potentially autonomous nature of the node wrapper operations it is a compromise that is required for supporting game development specifically in favor of a purely generic system.

**INodeDescription**

Each implemented node wrapper the module developer creates a description object using the INodeDescription interface. This is used to provide the system with a definition of what type of wrapper each one is, what inputs and outputs it expects as well as its name and a short description about its functionality.

**INodeRegistration**

The node registration interface is similar to the stream registration in that it is requested and used by the core system. However the information it provides is about the
structure of each available node wrapper through their associated description object as well as managing the instantiation.

5.1.2 Runtime System Interfaces

As the central controlling point for management and processing of procedures the system interfaces are used to provide the developer’s application with the ability to load, retrieve and invoke procedures. It also exposes interfaces that get passed to all module implementations to provide access to data that is being passed too each node’s operation functions.

ISocket

Passing data streams to and from nodes within a procedure is facilitated through the socket interface. Data streams used sockets as both the transmission and recipient points for data being transmitted. While the interface is common for both these two controlling points there are two different types of sockets that can be distinguished by the exposed type identification operator. These types are input and output, which defines their actions upon receiving a data stream. An input socket will accept a data stream, then notify the stream interface that it has been additionally referenced and store the data stream interface for later access during operation of the sockets owning node. The output node type will simply accept a data stream but then forward on that stream to all the input sockets that the system has associated it with. This interface is designed to allow only a single data stream to be associated with a socket, this ensures that no two streams are competing to write to the same socket while still allowing a single socket’s data stream to be output and read by multiple input sockets. It also ensures that run time does not need to worry about explicitly being aware of where data came from or where its intended destination is, this provides the developers the chance to reuse nodes where it is necessary without a context to the higher level task it is a part of.

INode

The term Node is used in place of ‘process’ as it is commonly referred to within data flow design and as such it is one of the most significant part of the process in terms of
how the user treats and places it so due to this change it is referred to slightly differently. Nodes encapsulate a process via the external node wrapper interface and have inputs and outputs for receiving and sending information through socket interfaces. The node interface is directly implemented within the framework so that it can be directly controlled and managed within the library. It is through this interface that the node wrappers implemented in external modules are able to access the desired input data streams.

**IProcedure**

Similar to the node interface the procedure interface is implemented within the core system however it is exposed to the developer’s application. The procedure interface encapsulates the full complexity and management of a designed procedure and exposes only a few operations to the invoking application. Of the operations exposed the most important are the ability to retrieve inputs and outputs from the procedure and a method to allow the application to run it. The input retrieval allows for access to set initial parameters or dependent data streams can be passed to the procedure prior to the run method being invoked and the output access allows for the results of the procedure to be accessed. Running the procedure itself is managed within the system’s implementation however exposing the ability to directly invoke it allows the developer full discretion of ensuring it occurs when they want to, whether it be explicitly or they wish to delegate it to some encapsulation object of their own design.

**IProcedureGroup**

While not directly exposed to the application developers or external components the procedure group interface is still implicitly referenced when retrieving a procedure. A procedure group is simply a collection of procedures that are grouped into a common namespace. This allows the developer to create and organize their procedures with a higher level encapsulation to avoid task specific operations that may be similarly named or identified from clashing and ensure that referencing them can be done simply by adding an explicit reference to the containing group.

An example would be if the system contained two groups of procedures named ‘Image’ and ’Audio’ and within each there are procedures that are both called ’Save’. In
order to reference these procedures the requesting application would need to explicitly identify which group the procedure was a part of. These identifications are done using the '::' string and thus the 'Save' procedure within the 'Audio' group would be referred to with 'Audio::Save'. This referencing can be seen in use later in the usage of the system.

### 5.1.3 Editor System Interfaces

The editor system and its interfaces expand on the run time system by exposing far greater control over the internal components of the core ProcLib systems. These are designed to allow the further integration of the library into a developer’s pipeline by using it within their own design tools and being able to abstractly control nearly all aspects of a procedure and higher level groups. These systems afford modification and generation of procedures and groups as well as offering the ability to run the created procedures or save them to intermediary files.

**IEditableNode**

Editable nodes are extensions upon the run time node interface. One of the minor and obvious additions is the ability to alter the identification name of a node. The significance is not vastly encompassing however it does allow the designers to give descriptive or applicable names to nodes. This can become useful when multiple nodes of the same time are added to a procedure but serve different purposes.

The major additions to the editable node are the operations that allow the designers to add custom or extended inputs and outputs to the nodes. Not all implementations of the node wrappers will use these but those that do allow the designer to pass and output custom data on an instance specific basis. The extension feature provides the ability to further dynamically alter the procedure beyond even just the simple variations of the data passed through the system’s inputs or value of the data being used.

**IConnection**

The connection interface is a significant interface for the editor system and also unique. While the run time system exposes no information of which nodes and sockets are
connected to each other or of the general path that flow of the procedures follow. In the editor this information is explicitly available through the connection interface.

Through this interface the editor is able to identify and associate each connection that is made within the procedure. Though the connections are non modifiable directly by the editor they serve the purpose of allowing it to visualize or describe them to the developer so that they are aware of the current state of the structure that the data will follow.

**IEditableProcedure**

Much of the direct control that the editor provides over the procedure’s structures that the developers are designing comes from the editable procedure interface itself. This interface exposes the ability for designers to add and remove both inputs and outputs to the system to allow the calling objects to provide data in a manner the designer is able to use and direct into appropriate nodes within the process. This control for directing data comes from being able to define the creation of connections between input and output sockets that exist within the procedure as well as the ability to request the destruction of already created connections.

This interface is also the one through which the designer is able to request the creation, destruction and listing of nodes for the procedure. The creation process of a node is as simple as requesting the creation of it by name from the available modules, once it is added to the procedure it is wholly owned by it but the user is provided access to its interface but all manipulation and further references are made through the procedure using the nodes name.

**IEditableProcedureGroup**

As with the run time, the editor stores all procedures in named groups. It is the editable group interface that exposes these groups to the editor and provides the interactions. The primary features of the group are to allow for creation, destruction and location of procedures. The groups are instanced in one of two ways, the first being that it is loaded from an external file and made available for modification while the other is to be instantiated from scratch with which the editor application is able to generate all new procedures from the ground up.
5.1.4 Application

Though not required to be implemented there are interfaces that are designed to be used by the application that is using the ProcLib system. These all provide the role of call back objects and are all used purely in the editor mode of the library. These are designed to expose a more event driven feature set so that the editor applications are able to reactively handle event information about changes within or to the library rather than to have to pro actively poll systems or interfaces.

IEditableProcedureCallback

The procedure call back is registered by the application with an editable procedure and will notify it of any nodes that are modified, created or destroyed. It will also notify destruction and creation of connections providing both names for the source and destination nodes as well as the specific sockets that the connection joins. This interface also implements notifications for the addition or removal of inputs and outputs to the procedure itself which can be used by the editor to confirm that requested additions or removals did in fact occur.

IEditableProcedureGroupCallback

Editable groups also can accept the registration of call backs using this interface. What the editable procedure callback allows the application to process is the creation and removal of editable procedures from a specific group. This ensures that the editors internal listings of associations with procedures that it may be keeping can be updated as needed. The callback provides notification of any editable procedure interfaces that may also have been created during the process.

IEditorCallback

The editor callback is the highest level callback for an editor application, it is registered at the creation of the ProcLibEditor. It allows for notification or confirmation of editable procedure requests, destructions and when they are loaded from file.

The editor callback also provides module load notifications and passes the node description interface of that module along with the referencing name. This allows the
editor to list all the available nodes as well as associate them with the higher level module title for further specification when requesting creation.

5.2 Procedure Implementation

5.2.1 Structure

The general implementation components of a procedure are very simple and follow the outlined structure from the design with very few extra objects. A full procedure is simply comprised of

- Sockets
- Nodes
- Inputs
- Outputs
- Connections
- Procedure
- Procedure Group

The procedure class itself is made up of collections of the core implementation of each of these. They are all internally implemented within the library to provide the most direct access for organizing and ensuring their operational structures.

CSocket

The internal socket is almost a direct implementation of the socket interface. The only addition that it offers is being able to directly reference other socket classes. This comes in the form of methods that store direct references to other sockets, which are then used for input type sockets to forward on all write commands to those connections without having to reference or store the data stream itself.
CNode

The node implementation uses both the editable and normal runtime node interfaces. When a procedure creates a node the constructor for the node manages the creation of all its required details. This is required because the node implementation is common and the only variations between the nodes directly are the required inputs and outputs as well as the associated INodeWrapper interface. The procedure creating the node passes it a reference to an instance of the associated wrapper interface that has been requested. Using this node wrapper interface it manages the instantiating of input sockets and outputs sockets based on the description provided by the developer of the wrapper node. It also sets the values of any parameters that might be passed for that specific to the instance of the node. The node retains the wrapper reference so that it can refer to it both when the node is require to run as well as to ensure that the destruction even is called for the node wrapper when the node itself is destroyed.

CConnection

Connections between nodes are defined by the implementation class of the connection interface. The connection class itself contains references directly to both the destination node and source node as well as the specific sockets rather than relying fully on names to retrieve components. The connection is not used direct in the operation of the procedure, but rather they are all stored by the procedure and when one is changed, added or removed it parses the entire collection and uses them to set up the direct connections between sockets. This allows connections to retain an element of speed during the operation of the procedure as all connections are direct but for the editor the connections can be modified and altered which will then result in the faster connection types being rebuilt based on the data from the connection classes.

Inputs

The system inputs are implemented simple as sockets and associated with a special node name within the connections list, specifically ‘input‘. They are stored by keeping a reference to the socket instance and the name it is associated with. Beyond that inputs are treated like any other socket in the procedure.
Outputs

Similar to the inputs, the outputs are simply sockets that have no association to a node but are referenced in connections using the node name ‘output’. Though referred to as outputs these sockets are actually an input type. This is due to them referencing from the external point of view from the procedure. Internal to the procedure they are written into from output nodes yet when the procedure is complete the output or results are stored within these outputs.

CProcedure

The implementation container for the procedure itself is both a manager container and operation unit. Implementing both the runtime and editor procedure interfaces it maintains the current state of a procedure and all its associated component instances.

On top of the organization of these components it also implements the run procedures of the interfaces. This is done using a fairly simplistic algorithm that could potentially be extended up on later. It simply iterates over all nodes that have not been run yet, as it iterates over each it checks if the required inputs have been written to. If the node is ready to run it invokes its processing operation and waits for that to complete before continuing. After it reaches the last available node it repeats from the beginning. When there are no more nodes left unprocessed it will finish up its operation and return. If it detects that there is a dead lock or for some reason there are nodes that cannot run then it will notify the application and abort the operation.

The operation of the procedure is simple but it is possible to suppress warnings to allow branching which is likely to leave nodes that have not been operated. However for the sake of the scope of the project and time constraints other, possibly more analytical and context aware, solutions have been left unimplemented.

CProcedureGroup

The implementation of the procedure group interfaces is very straight forward. It implements the full functionality of each and by using CProcedure objects it is able to manage both runtime and editable procedures.
5.2.2 Storage

Along with the implementation of the procedures came the need to define a storage format to allow designers to export their creations for import later. This file was originally intended to be custom defined to minimize space requirements however during the implementation it became obvious that a more flexible format would be structurally superior. The formatting of XML was found to be ideal for the situation as it was for more human readable but also the structuring was much more apparent. While the editor library is capable of generating the output file it was also considered that developers might end up creating the files themselves for some reason so the XML structure would allow them to easily replicate and organize these files.

The resulting file format can be viewed in figure 5.1. This file structure closely resembles the logical structure of the procedure’s and their groupings.

In brief the structural components of XML that this file takes advantage of are elements, the ability to embed elements and attributes. There are much more complex features available to XML but none of them are used.

The highest level element is the 'proceduregroup' which has a single attribute titled 'name' which is used to identify the define and identify an actual procedure group object. Within the procedure group element are multiple child 'procedure' elements. As would be expected the procedure element represents the procedure object created within the system and identified by the value of a name attribute.

Within the procedure element is a series of child elements. The first of these is the 'module' element. This is used to identify the modules that are required by the procedure. Each individual module is contained in a 'module' sub element by providing a title for the module and a path to the file that is to be loaded with the 'name' and 'file' attributes respectively.

The following two elements are representative of the input and output sockets for the procedure. Using these, the procedure is able to generate the appropriately named sockets for both access within the procedure and externally prior to being run. Sub elements in each titled 'parameter' identify an individual socket and the referencing name of each socket is provided through the 'name' attribute.

Following the inputs and outputs is the 'nodes' definition element. Within this is a collection of 'node' sub elements. Contained within each node element are attributes to
Table 5.1: ProcedureFileFormat
identify a node object’s name and type. The name attribute is a typical identification element but the type is an example of another scoped reference. The structure of the type identifier is a string composed of the title of the module to look in, followed by a double colon, which is then concatenated with the type declaration from within the module. An example of this usage would be to load a module that contains a node called ‘Translate’ and within the module declaration give the module the title ’Maths’. The type declaration required to instance a Translate node from the Maths module would then require the attribute to read ’type=”Maths::Translate”’.

Within the ’node’ element are sub elements each identifying components to be added to the node object when it is loaded. The elements are ’parameter’, ’input’ and ’output’. A ’parameter’ element is used to provide a default value to a named parameter that the associated node wrapper expects. The parameter can be used to act as a constant value that is varied with different instances which is passed to the node when it is run but without the need to associate the input through a socket. Extra ’input’ and ’output’ elements within the ’node’ element represent additional extended inputs and outputs, respectively, for the node object. These are identified with the ’name’ attribute and created when the node is loaded so that they can be used exactly the same as other sockets.

The final child element within the ’procedure’ element is titled ’connections’. This element contains all of the definitions for the connections that exist within the current procedure. Each ’connection’ child element is made up of two sub elements. These sub elements are the ’to’ and ’from’ components of a connection. They represent the source and destination information which is made up of attributes that identify the ’node’ and ’socket’ names to connect.

By using this file format and structure the system is able to encapsulate all required information to recreate procedures and ensure the appropriate modules that the nodes need have been loaded.

5.3 System Implementation

The final component in the implementation of the library were the interfaces that would be exported for use by the external applications and the management system for handling the loading of both procedures and modules. As they were to be the initial
and principal interaction unit between the library and the external application a large emphasis was placed upon ensuring clear and concise interaction between the systems.

5.3.1 Outline

An initial diagram of the runtime component of the system showing the sequence planned for loading a procedure and then the application running that procedure can be found in figure A.1 in Appendix A.

While the editor system itself is more complex with its interactions with the procedures themselves, the only big difference between the runtime interface and the exported editor interface is the ability to request the creation and saving of procedure groups rather than just the loading of them.

These exported interfaces that the developer application would need to interact with are the CProcLib class and the CProcLibEditor class. Both of these classes bridge the internal implementation of the library, CProcLibImpl, and the features that each type of application would require.

The structure of this interaction with applications, though differing on needs could reuse common concrete classes.

5.3.2 Components

CParser

The parser class is a utility class that uses the TinyXML library to read and save the XML files associated with the project.

During the loading of a procedure it iterates over each procedure group element instancing a new CProcedureGroup object. It then registers the group object with the CProcLibImpl class, mentioned later, that called it. It then builds the internal structure of the procedure, requesting modules to be loaded if not already loaded into the system and instancing CNode objects and CConnection objects as it finds new node and connection XML elements. As each new object is created the parser notifies any registered callback functions of the events. Upon successful completion of each procedure group the parser will invoke the initialization method of the group which propagates notification of the completed parsing to all the group’s sub components.
When the parser is requested to save a procedure group it performs almost the reverse of the loading process. Using the existing group object it builds up the XML output file based on the previously noted formatting and then writes that file to disk.

CModule

In order to allow external modules to easily interface with the core ProcLib system the CModule class encapsulates the operations required to load a module as well as retrieve the appropriate interfaces.

Encapsulating and abstracting the operating system specific requirements to load a module allows for each system module to be treated as a simple abstract component. Each module object represents a single module file, under Windows these are Dynamic Link Libraries, and exposes functionality to retrieve the INodeRegistration and IStreamRegistration interfaces that are exposed by those module files.

The CModule class also handles the unlinking and shutdown processes required by each specific operating system.

CModuleManager

With many different modules potentially being loaded the CModuleManager is another utility class, however it is used to simple manage creating and referencing CModule objects. There may exist situations where a single physical module file is register

CProcLibImpl

CProcLibImpl is the key stone to the entire ProcLib system implementation. This class encapsulates all the management of procedure groups as well as being the owning class of the module manager and the parser classes. It is through the CProcLibImpl that requests to save, create and load procedures and procedure groups get handled whether it be delegated to the parser or instantiating the appropriate objects. This class also handles the requests or requirements for loading modules which it does by invoking the appropriate operations through its’ module manager object. The CProcLibImpl also invokes the callback for major events such as the addition or removal of modules and the events associated with procedure groups including creation, modification and destruction.
CProcLib

The primary class that a runtime application interacts with is the CProcLib which bridges the core library implementation object through its internal instance of the CProcLibImpl class. Once an application has created or loaded an instance of the CProcLib it is able to use the available methods to request the loading of procedure definition files and request procedures by name.

The CProcLib only exposes these two features to ensure that the procedures and library won’t be modified once they have been created. Even though it uses a CProcLibImpl object to manage the functionality of the library it uses the multiple inheritance of the implemented classes to ensure only runtime interfaces are exposed.

CProcLibEditor

Very similar to the CProcLib class, the CProcLibEditor class owns an instance of the CProcLibImpl class. However the editor offers further features to the developers application. These are the ability to request the loading and destruction of specific modules by their filename as well as explicitly requesting the creation, loading, saving and destruction of editable procedure groups. Again using inheritance the editor class ensures that any interface to an object that is passed to the application is the editable version. The CProcedureEditor class also ensures that if the application wishes to register a callback with the system that it is appropriately passed to the CProcLibImpl object for use.
Chapter 6

Evaluation

Throughout the scope of the project it was always intended as an exploration of the feasibility of creating a library that could allow for integration of scripted procedural content. To this extent there was no empirical analysis of performance there were however several proof of concepts developed for implementing and testing various components of the system.

6.1 Tools

Currently many developers are using RAD based features of languages such as C# in the Microsoft Windows operating system in order to quickly build their in house tools. While these tools are able to provide fast creation of Graphic User Interfaces they often do not natively support the actual language or modules that the game developer’s technology is created in. Therefore the tools creators are faced with either creating simulations of the technology in the tools native language or creating wrappers that allow the native technology to be accessed from the tool.

The wrapper approach was used for the development of demonstration tools due to high importance placed upon developing a system that can be integrated into the developers tool chain. As the editor functionality was already built into the library and it handles all the features required to load modules, create and save procedures it made sense to only require a light Common Language Runtime (CLR) wrapper module. This also ensures that the library the designers are working with is functionally
identical to one that the runtime technology uses.

6.1.1 Managed Code Wrapper

The CLR wrapper for the ProcLib library is implemented as a managed Dynamic Link Library under Windows and referred to as ProcLibManaged. Intended purely as wrapper it uses heavily the software design pattern called a bridge[11]. A bridge, simply put, provides an abstract interface class that internally retains a reference to an implementation object. Using a C++/CLI compiled library the ProcLibManaged system is able to declare interface classes that can be instantiated or referenced by C# code but internally within the interface classes they reference a native C++ interface that exists within the ProcLib library. Thus essentially bridging the C# object with the C++ or native library object and allowing them to be treated as one and the same.

The core component of the ProcLibManaged library is a simple CLI template class that manages the storage and referencing of a provided native C++ interface. The source code for which can be found in C.1 in Appendix C. It is an unimpressive class but acts as the foundation class for all higher level interfaces used by the ProcLibManaged code. Figure C.2 in Appendix C provides an example of the CLR declaration code written to allow for interfacing with the native interface created within the library for a runtime version of a procedure. It exemplifies the simplicity of the CLI wrapper classes. By extending the native wrapper template with a specific interface explicitly ensures that the system will provide that type of implemented object on construction as well as certifying that the native interface will be available for the wrapping class. All that is left for the wrapper to do is implement the proper interfacing methods and pass them through to the associated native interface. Figure C.3 in Appendix C again shows the code required for the Procedure wrapper however it lists the code used to implement the declared methods, especially with respect to passing CLR invoked methods to the appropriate native method.

Implementing CLR wrappers for all the required exposed interfaces of the native ProcLib became an almost trivial process after the initial design plan was implemented. There was however a final concern associated with the wrapper system. This came in the form of events and callback interfaces. While the native callback interfaces were sufficient for the native code it would not be as simple to allow the library to call back
into managed code. Again though bridging offered a solution to the issue. Internal to the ProcLibManaged module simple classes were created that implemented all the available callback interfaces. These CLR callback objects each contained an association or reference to a managed CLR object so that when the native callback was invoked it would pass the messages back to the managed object. This allowed for the CLR interfacing classes to expose event delegation systems that would be familiar to C# developers while internally they would actually instantiate a native C++ object that would be registered as the native call back but still ensure the managed call back received messages properly.

6.1.2 C# Based Editor

With the ProcLibManaged module completed the first application implemented that would utilize its features was a simple C# based editor. The editor was intended to be a standalone unit as it is a proof of concept for using data flow visualization to allow a designer to build procedures by interfacing with the ProcLib library through its ProcLibEditor interface component.

There were three main visual requirements for the editor. The first of which was the ability to list all the currently loaded modules as well as the nodes they contained that could be used in procedures. The second was to similarly list all the loaded procedures within the library that could be edited. The final visual element that was desired was a work space in which to visualize individual procedures and their components. These were broken down into the sockets, nodes and connections each receiving a unique visual representation.

These visual elements also required appropriate interaction for the user of the editor, however it was also intended that the editor itself minimize the actual amount of processing of the user actions as possible. The editor was to function purely as a go between, so to speak, for the user and the ProcLib library. The design for the editor called for using the visual elements or extra graphical elements to allow the user to request actions and then the editor itself would simply collect all implicit data required by the library for that event and invoke it. Using this event system the editor’s internal state would not be changed at all until told to do so through callback events.

An example of this event driven interaction is for requesting the library to load a
new module. The user action to do so would cause the editor to display an information box such as in figure 6.1, which would be used to gather the desired module file and name to reference it with. The editor then sends a request to the load module method of the ProcLibEditor interface. The information is not used by the editor to populate or modify its current module listings; it simply passes the request on. It is a registered callback that manages the confirmation of a successful load for the desired module. Through this callback not only is the modules reference name available but it also provides the editor with the desired interfaces for listing all available node types.

Using this event driven series of requests and confirmation the editor is able to run all features required by the developer to create procedures. With the module loading already mentioned, the procedure and procedure group creation and management was similarly handled. If the user wished to create a new procedure they used a similar input box that took the name of the new procedure and the group to associate it with. The group association was implemented using a drop down box to allow the user to select an existing group or manually type in a title for a new group. Again when they completed filling in the required data the editor would pass on the request to the ProcLibEditor. Then if the creation was successful the appropriate events would be fired that would allow the editor to know when a new group was created and when a new procedure was added to a group.

Using this system of callbacks the library is able to notify the editor of nearly all important state changes through an extensive list of available events, as can be seen in table 6.1.

The editor was not entirely a messaging system between itself and the ProcLib library, did have to implement logic for allowing the user to manipulate the visual aspects of their created procedures. As the visualization is left up to the editor, the
<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProcLibEditor</td>
<td>OnModuleCreated</td>
</tr>
<tr>
<td></td>
<td>OnModuleRemoved</td>
</tr>
<tr>
<td></td>
<td>OnProcedureGroupCreated</td>
</tr>
<tr>
<td></td>
<td>OnProcedureGroupRemoved</td>
</tr>
<tr>
<td>EditableProcedureGroup</td>
<td>OnProcedureCreated</td>
</tr>
<tr>
<td></td>
<td>OnProcedureRemoved</td>
</tr>
<tr>
<td>EditableProcedure</td>
<td>OnNodeCreated</td>
</tr>
<tr>
<td></td>
<td>OnNodeRemoved</td>
</tr>
<tr>
<td></td>
<td>OnConnectionCreated</td>
</tr>
<tr>
<td></td>
<td>OnConnectionRemoved</td>
</tr>
<tr>
<td></td>
<td>OnInputCreated</td>
</tr>
<tr>
<td></td>
<td>OnInputRemoved</td>
</tr>
<tr>
<td></td>
<td>OnOutputCreated</td>
</tr>
<tr>
<td></td>
<td>OnOutputRemoved</td>
</tr>
<tr>
<td></td>
<td>OnOutputRemoved</td>
</tr>
</tbody>
</table>

Table 6.1: Listing of Editor Callback Events

layout of procedures was of particular interest, especially when it came to showing similarities to some existing systems that developers would be familiar with. Using the ProcLib’s data flow was the starting point for the visual, it used simple rectangular components to represent nodes with the name of the node at the top and down each side the input and output sockets. By using the left side to list inputs and the right side for outputs the editor discretely provided the user with a directional flow, which is commonly found in node based editors. The input and output elements of the editor are represented as special case nodes that work as marking starting and end points for the flow of the procedure. The connections created between sockets were represented as lines starting at the source socket and terminating at the destination socket.

Figure 6.2 shows the implemented editor in its full realization. With the graphical representation of the procedures in place the editor allows users to ‘grab’ nodes and move them around the work area to help organize it in a way they are happy with. In order to add a new node to a procedure they simply select the appropriate node on the list of available modules or libraries and then select the ‘Add Node’ button. Once a node is added to a procedure they are able to connect its sockets to other nodes by simply clicking on the desired source socket and then selecting the corresponding destination socket of their choice. The user also has the ability to destroy connections
and remove nodes through the use of selecting the object and clicking the appropriate removal button. When working in a procedure they are also able to add and remove inputs and outputs by selecting the appropriate buttons on the input and output nodes. If they wish to switch to a different procedure to edit they simply select the procedure in the list of available procedures and the workspace will update to reflect the desired procedure. Through the file menu the user is able to access all the loading, saving and creation events for procedures, procedure groups and modules, as well as general application events.

The true value in a demonstration application like this is to showcase the flexibility of the library to integrate into a developer’s tool chain with an effective and thorough event system to maintain consistency between the tools’ states and the internal library state. It also represents the similarities between existing node based editors and concepts that developers are using and the structuring of the generic procedure systems.
6.2 Modules

To evaluate whether or not the framework is able to use designer generated procedures to create content there needed to be modules implemented with which to test. These modules were designed around potential data types that a developer would be using within their game as well as some fairly standard data types that would be expected to be associated with procedural content requirements. Due to time constraints these modules certainly are not feature complete and in may not be representative of how game developers would implement similar features.

6.2.1 Core

The Core module was developed to provide both utility features and manage operations that would not be directly associated with any specific topic. These operations are mostly centered around mathematics and include simple arithmetic operations such as taking in two integer streams and adding their contents together to result in a single output data stream. It also includes mathematical functions like sin and cos, both of which take floating point values and return the corresponding value when passed through a sin or cos function respectively. The core module also contains constant value generators which simply output a data stream containing a specified value, on top of the constant value generators it also has a time generator which outputs a floating point representation of the running time.

The utility functions that the core library provides are mainly focused on recursion. These features include a for loop that iterates a specified number of times invoking a desired procedure with provided information and the iteration count. There is also a for each loop that is implemented, this loop parses a provided data stream as if it were an array, extracting each element one at a time and providing it as input into another procedure which it then runs. The final recursion based node that this module implements is a procedure call node, all it simply does is takes an input stream and passes it to another procedure to use when it is run.
6.2.2 Image

An image oriented module was implemented to allow for incorporating image oriented features into procedures. There are several node types available from the module, of which there are two image creation nodes. The primary data streams generated by the creation nodes provide wrappers for image objects. The image objects provide interfacing to data that is created using the DevIL image management library. Some of the core features of the image nodes are the ability to load an image from a file, convert and image to an OpenGL compatible format and draw shapes on top of existing images.

6.2.3 Mesh

The final module implemented for the demonstration procedures is focused on meshes. There is a data generation node within the module that allows for creating an instance of mesh class for use within procedures. Other nodes are implemented to use the mesh data stream to populate the mesh with appropriate information. One of these nodes is the OBJ loader which loads the mesh data from files using the Wavefront OBJ format. Other nodes are able to generate shapes such as boxes as well as being able to save meshes to file.

6.3 Demonstration Applications

All demonstration applications developed are written in C++ and access the runtime, CProcLib, interface. The procedures are also all loaded through files and use only the previously described example modules.

The scripts that define all of the following procedures can be found in Appendix B.

6.3.1 Sine Wave

One of the simplest demonstrations is the sine wave test. It uses the Core module to retrieve the current time and passes it through a sine function. Figure 6.3 shows the general flow of the procedure when the demonstration application invokes it. Using the ‘value’ output socket from the procedure the result of the sine function is then
multiplied by two and the results are printed to the user. As seen in figure 6.4 this function and be iteratively run and the output value progresses predictably according to the time progression of the sine wave.

6.3.2 Texture Loading

The texture loading demonstration procedure is built using nodes from the image module. Figure 6.5 shows the visualization of the procedure. The input to the system is a stream containing a string which represents the filename that the invoking program wishes to load. By simply using that string as input to the image loading node, it is able to load many different formats and passes back a data stream with an image interface. That image can then be used as the input to an OpenGL conversion node which handles creating an OpenGL compatible image out of the input. This converted image is passed to the result as an integer identifier which is then available for the calling application to retrieve and use to render the image as figure 6.6 shows.

6.3.3 Mesh Loading

This procedure was developed using the model module. By taking a filename string that feeds the input of the OBJ model, which is also connected to a mesh generator, it uses
Figure 6.4: Sine Wave Results

Figure 6.5: Image Loader Diagram
these inputs to load the appropriate mesh and is returned to the calling application.

This demonstration also shows the Interface feature that was added to the structure of procedures. As can be seen in figure 6.7 the mesh loading procedure interacts with the demonstration application through an interface. This interface in particular is a material system that defines features to allow for loading and retrieving named materials. The interface is implemented within the calling application and a reference to an instance of that implementation is passed to the procedure loading system. When the OBJ Importing node is instantiated it receives this reference and can proceed to use it.

The purpose of the material system interface is to represent an actual system a developer might use in their game engine that may require direct usage or simply need to exist as a single instance.

In the case of the demonstration it is used to allow the OBJ importer to request the loading of materials that are associated with the mesh that is being loaded. Using a persistent manager for this type of feature prevents a file being loaded multiple times and gives the whole application easy access to stored textures.

As well as the interface feature the material manager exhibits an reusable feature
from the previously mentioned texturing demo. Internally the material manager uses the image loading procedure to load requested images. This is an example of using procedures in various different aspects of an application.

Figure 6.8 the results of the demonstration application being run using a simple textured mesh.

### 6.3.4 Terrain Mesh Loading

Similar to the mesh loading procedure the terrain loading terrain generates a mesh based on an input file. However as can be seen in figure 6.9 the input file format is an image. Terrain meshes are often represented as simply two dimensional grids and as such the horizontal and lateral position is implicit. However the height of each point on the mesh can then be externally stored simply as an image, usually grayscale to save space. To load a mesh from a height map the procedure uses a node from the mesh module. This node takes in an image file containing scales of each point and then uses that to calculate the vertical offset of the vertex.

With terrain there is also a need for advanced texturing, this procedure handles the texturing process as well. Rather than use one large texture with all the data required it procedurally creates a unique texture for each terrain. It uses three loaded
base images that represent certain altitude values, as it iterates over each pixel in
the output texture it calculates the terrain’s height at that point and copies the pixel
color of the corresponding base texture at that height. The resulting texture is then
converted to a format ready to be rendered.

The results for this application can be found in figure 6.10 along with the image
used to import that terrain mesh.

6.3.5 Terrain Mesh Generation

The terrain generation procedure is very similar to the terrain import, as can be seen
from figure 6.11. However a large difference exists in the loading of the height map
image. The height map for this procedure is not loaded from file, it is generated
procedurally using Perlin noise. The procedure generates a blank image and then
iterates over each pixel in the image setting the color value based a value calculated
from a Perlin noise node. The image is then passed to the height map loading node
and the system continues identically to the terrain importer. The resulting output
from this application can be seen in figure 6.12 along with the image generated with
Figure 6.9: Terrain Importer Diagram

Figure 6.10: Terrain Importer Results
the Perlin noise.

6.3.6 Building Generation

The building generation process is one of the most complex procedures created using the prototype modules as can be seen in figure 6.13. It is broken up into two key components.

The first part is the generation of the building geometry. The construction of the building geometry is handled by a node from the model library that creates a box within a given mesh. The box node has inputs that allow for variable length, width and height values. Within the building generator these are connected to constrained random integer values. The random integers offer minimum and maximum constraints so that the values will always be limited to a desired range.
The other component of the building procedure is the texture generation. Each building receives a procedurally generated texture to cover its mesh. This texture starts off as a blank image which is passed into an iteration node. Within the procedure that the iteration node calls there is a recursive call to a node that draws colored boxes onto the image. This structuring of the procedure allows it to traverse both the width and the height of the image. The node that draws the box is a member of the image module, and takes an x position and y position passed to it by the iteration nodes as well as a color value. The color value is generated from another random integer node that feeds all three channels of the color component to give gray scale value between black and white.

The resulting geometry and texture are passed back via the outputs to the application. These are then stored for rendering to the screen, which can be seen in figure 6.14.

### 6.3.7 City Generation

An extension of the building generator is to use it to generate multiple objects. Ideally this would also involve differences between the buildings to create a non homogeneous appearance. By using its implicit variation this is as simple as just repeating the call to the procedure. As each building is generated its mesh and texture are saved to an application defined structure and given a transformed location. The transforms
Figure 6.13: Building Generation Diagram

Figure 6.14: Building Generation Results
in the demonstration are simply based on the total buildings and placing new ones at the end of a list however it could be possible to expand the system to use more intelligent placement. A limitation of using the single building procedure which only creates one type of building is that they all look fairly similar even with their variations. The results, as seen in figure 6.15, however show that as a prototype the results are promising of further potential.
Chapter 7

Conclusions and Future Work

7.1 Conclusions

The objective of this project was the devise and prototype a framework for allowing game developers to implement procedure driven features into their technology while retaining the flexibility the designers would need to control these systems outside of the core game engine systems. While full exploration of the topic could potentially result in years of research, the narrow focus on this project to simply defining a framework system that had a formal definition of a procedure and a way of processing those procedures that leaves that actual definition and creation of each component of the the procedure up to the developers own needs and implementation.

This narrow focus on organization and management has successfully allowed for the implementation of applications both for designer creation of procedures and interactive, real time application based usages of those procedures. These are felt to be successful foundational steps in opening up games to much more varied and thorough adoption procedural scripting to compliment their existing technology.

By using the the flow based design for defining the conceptual structure of procedures allows a flexible implementation into very visually oriented editors systems. The flexible editor system built into the library can potentially be integrated right into a developers own tool set or tools that they have source code access to. Even though the system uses the flow concept extensively the actual visualization and manipulation of the structure and content of procedures is still left up to the developer should they
wish to represent it in some other way or incorporate it directly into their own editing paradigm.

For the technical developers and programmers of game technology the library offers a small set of required interfaces that they can implement as they see fit. Through the implementation it was a goal to not force the developer to use any specific formatting of their data and algorithms but rather to make the system flexible enough to handle data with little regard to what it was and to simply ensure that data was trafficked as procedure designer intended. Through the externalization of procedure nodes, using modules, the goal of extensibility and generality was achieved successfully as well, the entire system is able grow with a developer and their needs while offering the opportunity to proceduralize future and current features due to the framework for doing so already being in place.

7.2 Future Work

Further work on this project would most likely focus on extending the prototype modules. Possibly looking into what types of nodes that game developers would most commonly need and implementing those to save the developer time. While the flexibility of the library is important it could be just as important to have a large set of varied examples to demonstrate to encourage developers to explore the possibility of incorporating such a system.

The most important extension to this project however would be examining operational performance, graph parallelization and the general operation of a procedure. With performance analysis much would still remain dependent on the node implementations but potentially there could be areas in which data flow analysis or intelligently probing the connection paths of complex procedures could benefit with better operating speeds. The structure and implementation of procedures resembles elements of networking systems and could benefit from exploiting features found in that area of study. Linearly separable operating paths or pipelining of data could allow for improved performance and possibly allow for distributed running of procedures. Potentially distributing these procedures could be on the micro level between threads, CPU cores or even CPUs, this distribution could be used to harness processing power available on processors such as the Cell and Xenon found in the Xbox360, both of which are
target platforms for many game developers. One of the most interesting features that could be expanded on is branching within procedures. While simple implementations of recursion have been implemented there is little in the way of structured and controlled separable processing paths. For some needs there may not exist any desire to introduce the divergence that can exist with branching however in other cases it could be that even simple ‘if’ and ‘switch’ statement nodes could not only reduce the amount of processing required but also provide much more flexibility in determining actions within a procedure.

7.3 Closing Thoughts

This exploration of making scripted procedures an integral part of game development could potentially be beneficial to all the target developer types. Designers could retain much more control over how the games they design look as well as work. Procedural systems could possibly ease the burden of programmers from writing game specific code and focus on simply creating robust and reusable modules and components for designers to use in the creation of the game specific features.

Game development projects are going to continue getting larger both virtually and in asset requirements and it is likely that procedural content will become an important part in populating those games. This however will require more designer control over the structure and operation of those procedures to ensure enjoyable experiences for players and this project outlines a framework that could potentially be one way to ensure that control. Growing with the developers needs by being as flexible as possible while still being able to expand and progress with the technological advances.
Appendix A

Diagrams
Figure A.1: Sequence for Loading and Running a Procedure
Figure A.2: Full Architecture Overview
Figure A.3: Client Overview

Client

ClientApplication

<< interface >>
IEditableProcedureGroupCallback
+NodeCreated(): void
+NodeRemoved(): void
+ConnectionCreated(): void
+ConnectionRemoved(): void
+InputCreated(): void
+InputRemoved(): void
+OutputCreated(): void
+OutputRemoved(): void

<< interface >>
IEditorCallback
+OnModuleCreated(): void
+OnModuleRemoved(): void
+OnProcedureGroupCreated(): void
+OnProcedureGroupRemoved(): void
Figure A.4: Module Overview

**Module Registration**

- **Interface:** INodeRegistration
  - GetNodeCount(): void
  - GetNodeDesc(): void

**Module Description**

- **Interface:** INodeDescription
  - GetName(): void
  - GetDescription(): void
  - GetNodeType(): void
  - GetInputCount(): void
  - GetInputName(): void
  - GetInputRequired(): void
  - GetOutputCount(): void
  - GetOutputName(): void
  - GetParameterCount(): void
  - GetParameterRequired(): void
  - CreateWrapper(): void

**Module Wrapper**

- **Interface:** INodeWrapper
  - SetNode(): void
  - ParseDefaultInput(): void
  - OnInputChanged(): void
  - OnInitialize(): void
  - OnRun(): void
  - OnDestroy(): void

**Stream Registration**

- **Interface:** IStreamRegistration
  - GetStreamCount(): void
  - GetStreamType(): void
  - Create(): void

**DataStream**

- **Interface:** IDataStream
  - Reference(): void
  - Release(): void
  - Clone(): IDataStream
  - Snatch(): IDataStream
  - GetType(): Integer
  - GetDataType(): Integer
  - isArray(): Boolean
  - GetElementCount(): Integer
  - GetFrom(): void
  - GetAt(): void
Figure A.5: ProcLib System Overview
Appendix B

Procedures

Developed procedures can be found on the enclosed disk under 'Bin ProcLib procedures'.
Appendix C

Code

Full implementation code can be found on the enclosed disk in the 'Code' folder.
namespace ProcLibManaged
{
    template<class T>
    public ref class NativeWrapper
    {
        public:
            NativeWrapper( T* p )
            {
                m_pNative = p;
            }
            T* GetNative( void )
            {
                return m_pNative;
            }
        protected:
            T* m_pNative;
    };
}

Table C.1: ProcLibManaged Native Wrapper

namespace ProcLibManaged
{
    public ref class Procedure : public NativeWrapper<proclib::IProcedure>
    {
        public:
            Procedure( proclib::IProcedure* proc );
            ~Procedure();
            void Run( void );
    };
}

Table C.2: ProcLibManaged Procedure Wrapper
using namespace ProcLib;
using namespace ProcLibManaged;
Procedure::Procedure( IProcedure* proc )
    : NativeWrapper( proc )
{
}

Procedure::~Procedure()
{
}

void Procedure::Run( void )
{
    m_pNative->Run();
}

Table C.3: ProcLibManaged Implemented Procedure Wrapper
Bibliography


[37] I.D. Visualization. SpeedTree.
