

To the Graduate Faculty:

I am submitting herewith a project written by William Wayne Reynard Jr. entitled “An Artist Centric Particle Editor for Real-Time FX Visualization” I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Interactive Technology in Digital Game Development, with Specialization in Software Development.

Gary Brubaker, Faculty Supervisor

We have read this {keep one} Thesis/Project
and recommend its acceptance:

Accepted for the Faculty:

Executive Director, The Guildhall at SMU

AN ARTIST CENTRIC PARTICLE EDITOR FOR REAL-TIME FX VISUALIZATION

A Project Presented to the Graduate Faculty of
The Guildhall at Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Master of Interactive Technology

in Digital Game Development

with

Specialization in Software Development

by

William Wayne Reynard Jr.

(BS, Wichita State University, 2005)

March 21, 2007

Reynard Jr., William W.

BS, Wichita State University, 2005

An Artist Centric Particle Editor for Real-Time FX Visualization

Supervisor: Gary Brubaker

Master of Interactive Technology degree conferred March 23, 2007

Project completed March 21, 2007

Particle systems are used extensively in video games to provide user feedback and for general visual enhancement. The effects are commonly used to represent volumetric effects such as smoke or fire as well as effects with no real-world counterpart, such as spell effects. The application developed for this project is a tool designed to promote the rapid development of particle effects. It is designed to be familiar to users of 3D modeling packages such as *Discreet's 3D Studio Max*, and provides visual interactive controls for effect editing. Discussed herein are the issues encountered during the development of the tool and the usability results gathered from student artists.

ACKNOWLEDGEMENTS

I want to thank my masters committee for their suggestions and expertise without which this project would not have been possible. I also want to thank my parents for their immense support throughout my life, giving me an opportunity to pursue my goals.

My thanks and praise also goes to my fellow Cohort 5 programmers with whom I shared the trials of the past 21 months, and who I could count on for a fresh perspective on many problems.

An Artist Centric Particle Editor for Real-Time FX Visualization

Supervisor: Professor Brubaker, Gary

Master of Interactive Technology degree conferred March 23, 2007

Project completed March 21, 2007

This thesis explores the development of an Artist friendly particle editing program for use in the development of real-time particle effects for video games. Particle tools are typically developed by programmers. Preferably this would be the same programmer who developed the particle simulation back-end since they will be most familiar with the inner workings of the system. The programmer will likely never use the tool in the production environment or for any purpose outside of testing or demonstration. It is easy in this situation to develop a tool from the perspective of the developer and simply expose all control parameters to the user interface. This approach allows for all parameters to be modified, but leaves the issue of understanding how and what a parameter actually does to the user. Although all available parameters are exposed to the designer, some combinations of parameters may simply not work together or features may be mutually exclusive. It would require knowledge of the inner workings of the system in order to make predictable changes to parameters. This process may not prove too terrible if the number of parameters is low and the turn around time from editing parameters to visualizing the change is real time.

With some planning it is possible to develop a tool that artists find comfortable and intuitive to use. Cutting down on artist frustration will increase the number of times they are willing to iterate the effect therefore producing a higher quality end result. The approach taken in this thesis was first to determine what was most and least efficient about creating particle systems with current particle editors. Second the feature set needed to be determined so the interface could be designed with the correct data set in mind. Next it had to be decided how control parameters would be represented in the user interface and how they would map to simulation ready data. The final task was laying out the user interface and designing effective controls for the manipulation of data.

Because the user interface is a major focus of this study it is important that iterating the design be as painless as possible. To this end all UI development was done with Microsoft C# and Microsoft Visual Studio 2005. These technologies provided a suite of pre-built controls and the tools to develop new types of controls. The particle simulation code is completely written in C++ to provide a common back end to both the tool and the game. This code was accessible to C# via a C++/CLI proxy class. Particle rendering is accomplished through HLSL vertex and pixel shaders, which provide a common render path for the tool and game.

To determine if the tool successfully completed the goal of being artist friendly a set of usability tests were conducted with artists and level designers attending the Guildhall. The results of the tests are stated at the end of this thesis.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	IV
ABSTRACT	V
TABLE OF CONTENTS	VII
LIST OF FIGURES	IX
NOMENCLATURE	X
Chapter	
1. INTRODUCTION	1
1.1. Motivation	1
1.2. Particle Editor Goals	2
2. FIELD REVIEW	3
2.1. Use of Visual Effects	<u>4</u>
2.2. Visual Feedback	<u>4</u>
2.2.1 Weapon Trails	5
2.2.2 Weather Effects	5
2.3. Particle Editors	<u>6</u>
3. METHODOLOGY	<u>9</u>
3.1. Introduction	<u>9</u>
3.2. <u>Editor Application</u>	<u>10</u>
<u>3.2.1 PalmControls</u>	<u>11</u>
<u>3.3.1 PalmConnector</u>	<u>14</u>
<u>3.3.2 PalmEditor</u>	<u>16</u>
3.3. Particle Simulation	<u>16</u>
<u>3.3.1 Cache Efficiency</u>	<u>16</u>
<u>3.3.2 Stream Processing</u>	<u>17</u>
<u>3.3.3 Particle Motion</u>	<u>17</u>
<u>3.3.4 Integration Methods</u>	<u>18</u>
<u>3.3.5 Forces</u>	<u>20</u>
3.4. Particle Rendering	<u>21</u>

<u>3.4.1 Fill Rate</u>	<u>22</u>
4. RESULTS AND ANALYSIS.....	<u>23</u>
4.1. The Test	<u>23</u>
4.2. Results of Usability Rankings.....	<u>23</u>
4.3. Observed Usage Issues	<u>27</u>
4.4. Results.....	<u>28</u>
<u>4.5 Farther Work</u>	<u>28</u>
5. CONCLUSION.....	<u>30</u>
REFERENCES	<u>32</u>

LIST OF FIGURES

Figure	Page
2.1. 3D Studio Max User Interface	7
2.2 ParticleIllusion User Interface	8
3.1 PalmEditor User Interface – Iteration 2	9
3.2 PalmEditor User Interface – Iteration 1	12
3.1. RK4 vs. Forward Euler 60fps	19
3.2. RK4 vs. Forward Euler 30fps	20
4.1. Particle Editor Usability Rankings	24
4.2. Particle Editor Usage Statistics	26

NOMENCLATURE

Acronyms

UI	User Interface
GUI	Graphical User Interface
RK4	Rung-Kutta 4
AABB	Axis Aligned Bounding Box
AOS	Array of Structures
SOA	Structure of Arrays
MSIL	Microsoft Intermediate Language

Subscripts

i, j	array indices
--------	---------------

CHAPTER 1

INTRODUCTION

1.1 Motivation

Particle Systems play an integral role in video game applications. They are used extensively throughout the game to enhance visual appeal and often act as a visual cue to the player. Many times it is desirable for particle systems to be custom built into levels, or attached to a specific model. The motions of a particle over time are primarily described by the forces acting upon the particle, such as wind, gravity, or a spring. These forces are integrated over time during the execution of the program in order to determine the current position of the particle.

Who is responsible for describing the forces that will act upon a particle? Programmers are one obvious choice, and it is entirely possible that a game could be created using only hand written particle systems. The problem here becomes the difficulty in determining how the particle system will look when it's complete. Artists or designers will often want to tweak the effect to look just right, and having to have a programmer on hand just to tweak values is a waste of time, and in turn money. Adding to the waste is the issue of having to recompile each time a value is tweaked. Depending on how many times simple control parameters need to be tweaked this can add up to a huge time sink. Real time controls allow the designer to immediately see the affects of their parameter changes, allowing them to make more changes in the same amount of time. It can also provide a greater ability to understand how the parameters individually affect the resulting behavior by rapidly switching between potential values.

To eliminate compile time and not immediately knowing the result of parameter changes, one solution is to create a particle editing software package to generate a particle definition file. A naïve editor would allow for the user to type in force parameters that act upon a particle and display the resulting movements back immediately. This could be expanded to modify any particle property. Particle parameters common to all real time systems observed by the author include lifetime, position, velocity, acceleration, color, opacity, size, orientation, texture

coordinates, texture, and forces. Constraints to these parameters are sometimes available as well, such as positions or velocities relative to the particle emitter. Such an editor may be enough, and is certainly better than recompiling after every tweak to see results. The problem with such a simple editor is that it is not intuitive. This is especially true if it is being used by an artist or designer. They don't care what initial velocity vector is needed to get the effect to go the direction they want. They only care that it does indeed go the correct direction.

So what would be a better solution? It is important that the user be able to see how the changes they make affect the end result of the effect in real-time. It is also important that the controls available to the user allow them to edit effect properties in an intuitive manner. It is possible that the user of the program could be a game or level designer, or an artist which means that the user interface should expose controls that may be important to both. A technically minded designer may want to have very low level control over an effect, while an artist will be concerned with controlling parameters at a higher level. The editor for this project allows low level entry of explicit values and higher level graphical controls for all particle parameters.

1.2 Particle Editor Goals

Users of the tool should be able to easily modify the following control parameters to customize the look of the particle effect being designed. The user will be able to select an emitter shape from a predefined list which is provided by the particle simulation back end. More emitter shapes could be added very easily by a programmer. Common particle effect parameters that are editable in the tool include initial velocity with variation controls for initial speed and initial direction, particle color, size, texture, blending mode, energy (lifetime) with variation [Van der Burg, 2000]. All of these properties are editable when an emitter is selected in the scene.

Two other types of objects are selectable in the scene, gravity and attractors / anti-attractors. Gravity objects represent an acceleration to be applied to particles in a constant direction, much like how gravity is viewed on Earth. Attractors are point masses which create a force (acceleration) which is always in the direction toward the attractor from an individual particle. Attractors are actually more representative of actual gravity as thought about in more astronomical terms, such as how the Earth orbits the sun. Anti-attractors are attractors with negative mass, which reverse the direction of acceleration. These two controls were added in an

attempt to allow designers to control particle motion with forces observed in nature, albeit at a much smaller scale. In practice the control proved difficult to use for creating a specific path of motion. Since the force is always active the particles always moved in some type of orbit until their death, which limited the overall potential paths that could be created. The resulting paths were also very sensitive to initial condition changes, meaning slight changes to placement or mass could bring about surprisingly different paths of motion. The attractors did however create a very nice swarm of particles and could be easily used if that type of effect was desired.

Above providing any potential set of features to particle designers, this study was to explore what artists want from an editor when designing particle effects for real time simulations. The questionnaire given to the users of the program was intended to gather concrete evidence from the users about their experience working with this editor. All users who participated in the study were observed during the entire duration of their usage session in order to determine how they tried to use the program set in front of them and to witness their reactions to the user interface.

CHAPTER 2

FIELD REVIEW

2.1 The use of Visual Effects

The number of new video game titles released in a year is staggering. Because of this games now more than ever need to quickly and effectively show off their prowess and get players engaged as quickly as possible. Defining a fun experience is subjective to the player, but there are some common goals that games of all genres can work toward to increase player satisfaction. Immersion is the “suspension of disbelief” that players experience when completely enthralled in a game session. The games human interface devices (controllers, televisions, etc) cease to exist as far as the player is concerned, and they are left to happily exist in the game world. Creating this type of meaningful play is not easy, and conveying appropriate information without interrupting the players experience is difficult at best. Action interruptions, menu navigation, loading screens, these all interrupt the suspension of disbelief and remind the player that it’s only a game.

Central to an artists or designers ability to create stunning game content is the toolsets provided by the programming team. An excellent artist may find it difficult to produce any interesting asset of the tool they are required to use doesn’t allow them to work efficiently and effectively. The next generation of games is going to require high volumes of quality assets to be created by art teams approximately the same size as the previous generation [Screen digest].

2.2 Visual Feedback

A recent example of a game using visual cues to improve player feedback is Tecmo’s Ninja Gaiden. “Ryu's slashes seem to pack a tremendous wallop and leave enemies bloodied or even decapitated, which are effects that are far more stylish than gratuitous. Enemy attacks appear incredibly punishing, and indeed, they'll often bring Ryu to death's door.” (Kasavin, 2004)

When the player is attacking an impressive trail flows along behind the weapons attacking surface. These trails portray a real sense of speed and power behind every attack. If

the attack hits an enemy an additional impact effect is shown, along with appropriate sound effects. These cues provide a means to the player for determining the success of each attack. The same attack without the trails and visual impact notification would not convey the appropriate feeling of slicing through a foe. The attack would feel weak, useless, and worst of all boring.

Sony's God of War is another example of the use of visual cues. One particular spell available is turning enemies to stone with medusas head. When using the spell a cone of effect is displayed emanating from medusas decapitated head. Attacks in God of War also have trails added behind them, again to convey power and speed.

2.2.1 Weapon Trails

Weapon trails are an effective visual feedback system that portrays the power and speed of an attack. To create weapon trail particle effects the artist defines where on the weapon the trail will emanate from. Exactly how this information is stored is implementation specific but defining the emission location is likely to happen in the artists modeling program. When the weapon is used in game, the emission location will be tracked on the weapon as it animates. When it's necessary new vertices would be generated at the tracked locations on the weapon. All vertices are combined into a single geometric representation of the weapons swing trail and passed off to be drawn. Textures and alpha blends could be modified to fit the desired effect. Often times the vertices alpha value will decrease over time making them increasingly more translucent. When they become transparent they can be removed from the vertex list.

2.2.2 Weather Effects

Most games in the past have had a very static world that remains essentially the same over the duration of the game. For many games this is fine, since the relative elapsed time in the game may be very small. For other games this creates a very drab and unconvincing environment to exist in. Players of MMO games sometimes spend considerable amounts of time wandering around the game world. In a persistent world such as this the lack of changing weather becomes very obvious, and the player loses some sense of a real living world. One way to break up the monotony is to add in variable weather effects like rain or snow both of which can be effectively accomplished with particle effects. Lightning and clouds are other potential particle effects that could be made to increase the emersion of the scene.

2.3 Particle Editors

Creating visually stunning particle effects requires defining parameters such as size over time, color over time, rotation speed, acceleration, drag coefficients, textures and any animation of textures, collision surfaces, blending modes. Tweaking these values in code would require a recompile after every change, and can be extremely time consuming and aggravating. Configuration files allow the same tweaks to be made without recompiling, but you aren't generally going to be able to edit these with real-time affects. The particle effects for the Guildhall at SMU and Team Pants's *Half Life 2* total conversion *Grimoire* were created this way. A better solution is to provide a toolset for the particle designer. This allows the real-time editing and viewing of the particle systems, to ease creation. It is possible that the designer may be an artist, or programmer, or even a level designer. A tool to tweak variables doesn't provide a clear enough interface to how the changes will affect the system. A powerful editor would be one that freed the user from endlessly wading through lists of parameters just to find what they are looking for and empower the artist to work on the effect visually and focus their attention on the parameters they are using. The Unreal 2K4 particle editor does not expose all control parameters to the user via the visual editor, resulting in many effects having to be hand edited in scripts to get the proper effect. This is a frustrating way to work and keeps the user from viewing the final system in real-time. Another particle system editor that is available is a user created tool for the Ogre engine. Besides the lack of power that the engine seemed to have for particle processing, the editor simply accepted manual inputs of vector forces to be applied to the particles. This may work well enough for a programmer type who is familiar with coordinate spaces, but is less intuitive to a less technically oriented user.

Many particle editors allow for the customization, to some degree, of the emitters in the system. Fewer included the ability to load an arbitrary mesh, and to use its vertices as emitter positions.

[screenshots are "unreadable". Besides making them bigger, generally if you make screenshots for a document, its not a good idea to use your native 1920 res or whatever those are in. if you'd use 800 or 1024, the user interface elements would be much more visible.]

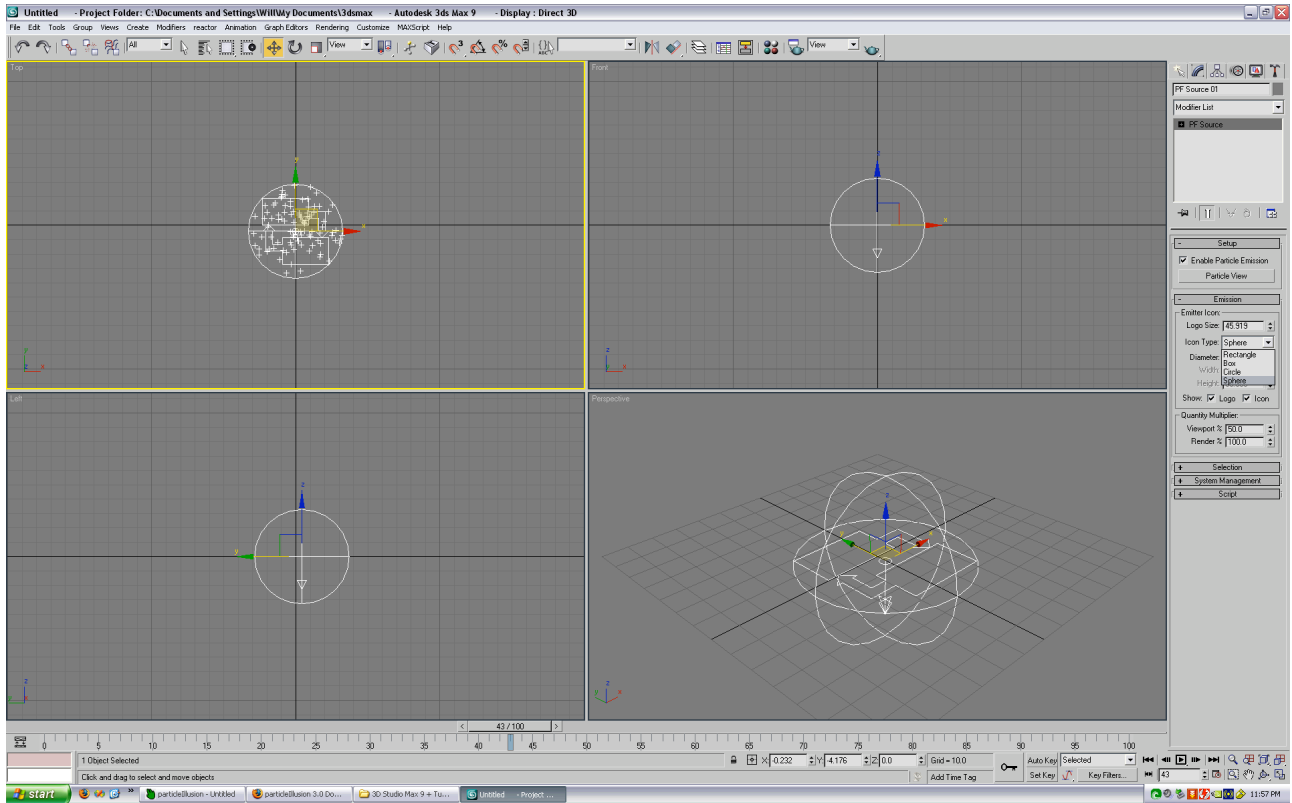


Figure 2.1 - From 3d Studio Max, on the right side you see a drop down showing a few selectable emitter shapes.

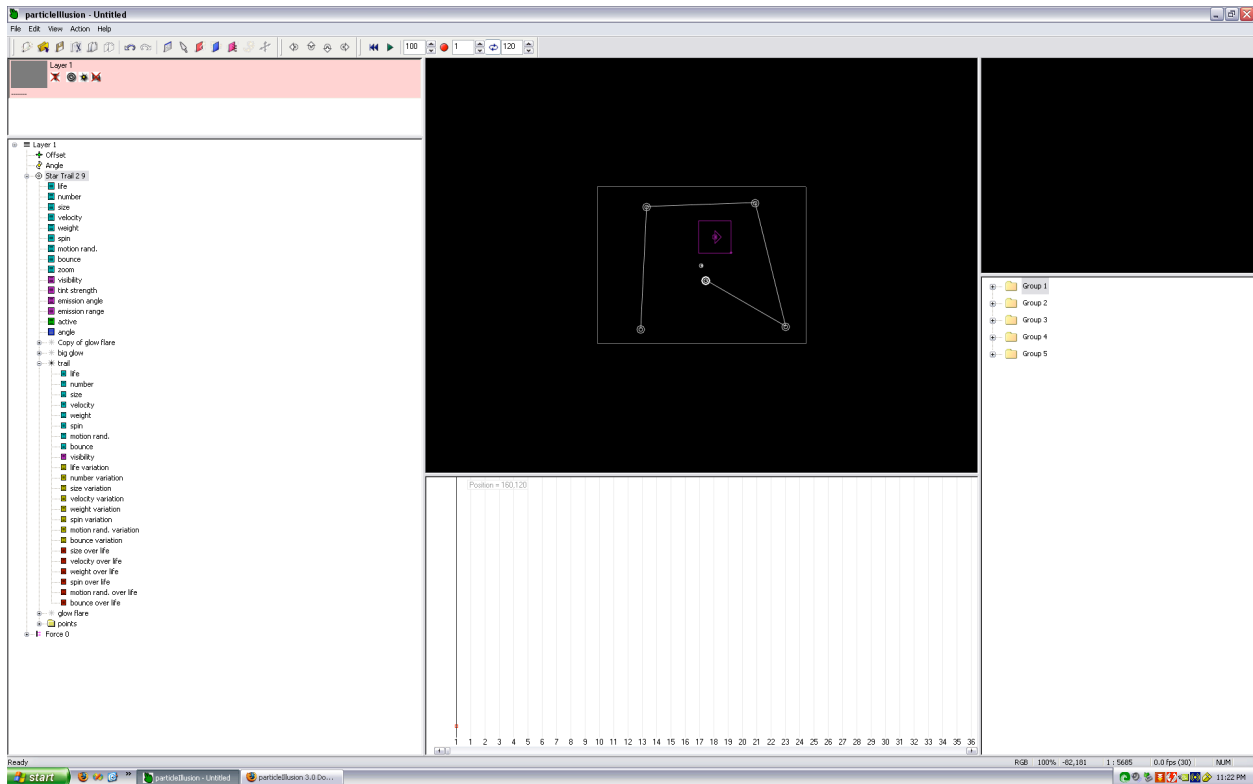


Figure 2.2 – This picture is from *ParticleIllusion*, multiple line segments can be connected and used to emit particles.

This is a good idea but it seems it could be taken farther. Allowing the user to load in an animated model, and defining groups of vertices to be emitters seems a much more flexible system. With this setup the user could define a region such as a hand, or the armor, or whatever, to become an emitter. Animating this alongside the character would produce the result of the specific region of the model apparently emitting the particles. Defining the affected areas should happen much the same way as artists select surface areas in modeling tools like *Maya*, which allows the artist to select individual vertices or faces with the mouse as well as selecting areas of a model with a mouse marquee. Such an affect could easily be used to have magic emitting from a wizards hand, or light an enemy ablaze with a flamethrower.

CHAPTER 3

METHODOLOGY

3.1 Introduction

This section seeks to explain the overall principles used when designing the particle simulation software and an artist friendly tool for design. The editing application was designed to work similar to 3d modeling programs, specifically *3d Studio Max*. Interactive controls were included to supplement direct parameter value entry in order to allow the designer to make adjustments easily in real time. The editor window has a preview window in the center, a tool bar on the left side for selecting common controls and adding objects to the scene. The right side of the application window contains a categorized control list.

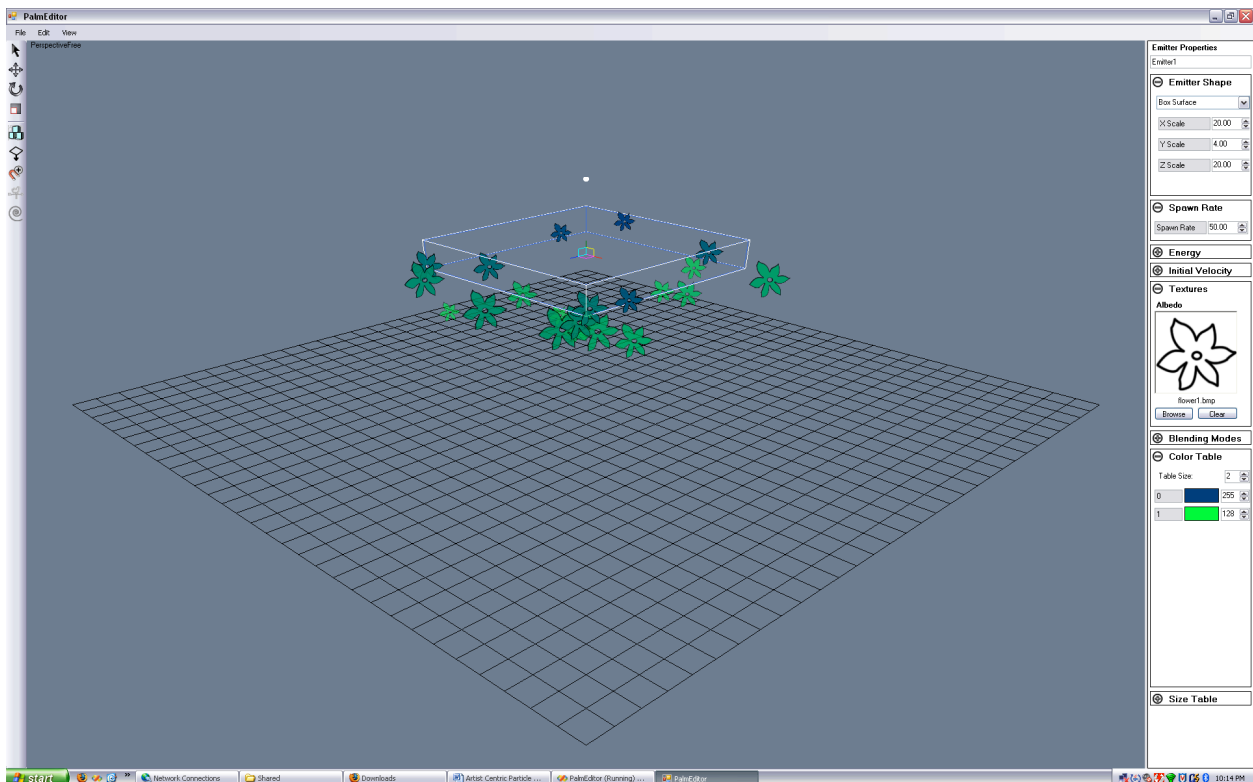


Figure 3.1 – The PalmEditor, common tools are available on the left. The main window is for previewing effects and direct manipulation of emitters and particle parameters. The right side contains all particle parameters in a collapsible list.

The particle simulation code was written as a set of C++ classes which form the core of the particle effect solution described in this paper, which has been named Chi (because it needed a name). Included is a class representing particle streams, which define the movement and rendering characteristics of a given particle system. Another class defines particle emitters, which determine the starting positions and spawn rates of particles. Finally there is a management class which handles the creation, deletion, and connections between streams and emitters. Additionally a 3D Vector and 3x3 Matrix class are included providing base math functionality.

3.2 Editor Application

The editor application and more specifically the user interface (UI) was the core focus of study for this project. It was important that the editor was comfortable and maybe even familiar to the user, who could be a designer, artist, or programmer. Since the particle designer seemed more likely to be an artist or someone with experience designing artistic content for games the UI was designed primarily for them. The basic 3D controls included in all 3d modeling packages are translation, rotation and scaling. For this application the basic controls were modeled after the *3D Studio Max* version of these same tools. *3D Studio Max* was selected as the modeling package to imitate because it is the primary modeling package for the artists who participated in this study. Providing consistency of tools is primary to the idea of minimizing the learning curve associated with a new application. The tool provides a set of editable properties on the right side of the screen which is again similar to the way *3D Studio Max* provides access to an object's properties.

C# was used to code the user interface because of its powerful suite of graphical user interface development tools. C# is packaged with a rich library to support the development of Windows applications and provides access to the Windows common controls. It also has an integrated visual UI for creating new graphical user interfaces. The editor application is divided into three major sections the PalmConnector, PalmControls, and PalmEditor projects. The Palm prefix to these projects exists only to identify them as belonging together. PalmConnector wraps unmanaged functionality and provides managed access to the unmanaged systems (in this case the simulation code.) PalmControls contains all the user interface code and PalmEditor represents the main editor itself.

The ultimate goal of the application is to enable artists to work on particle effects in an environment that is familiar and allows them to create on the computer the effect they imagine in their mind.

3.2.1 PalmControls

The PalmControls project is where most of the work for the editor was done. PalmControls contains the namespaces and classes for every editing feature used. An iterative approach was taken to develop the controls, meaning the controls were first built as quickly and easily as was necessary to get the correct behavior. After the first design was completed it was tested for usability, and modified as necessary. Finally the code was gone over to clean up and improve overall quality. The initial design of the user interface proved to be a trial by fire. There was no clear organization of where the user would need to look to find the properties they were interested in modifying. A particle effects' properties were split into separate windows accessible via tabs. Properties existed on either the emitter or stream page. Even if two properties seemed related they were shamelessly categorized exactly how they existed in code. Visual modification was also awkward, since the user would first have to find the desired property on the tab page, when it was selected a separate control would appear in the preview window which could then be interacted with via the mouse. Four preview windows were used showing the view from the top, left, front with orthographic projection and a perspective camera view. It was decided to collapse these views down into a single window that could switch between each view to maximize screen real-estate which artists at school seemed to prefer.

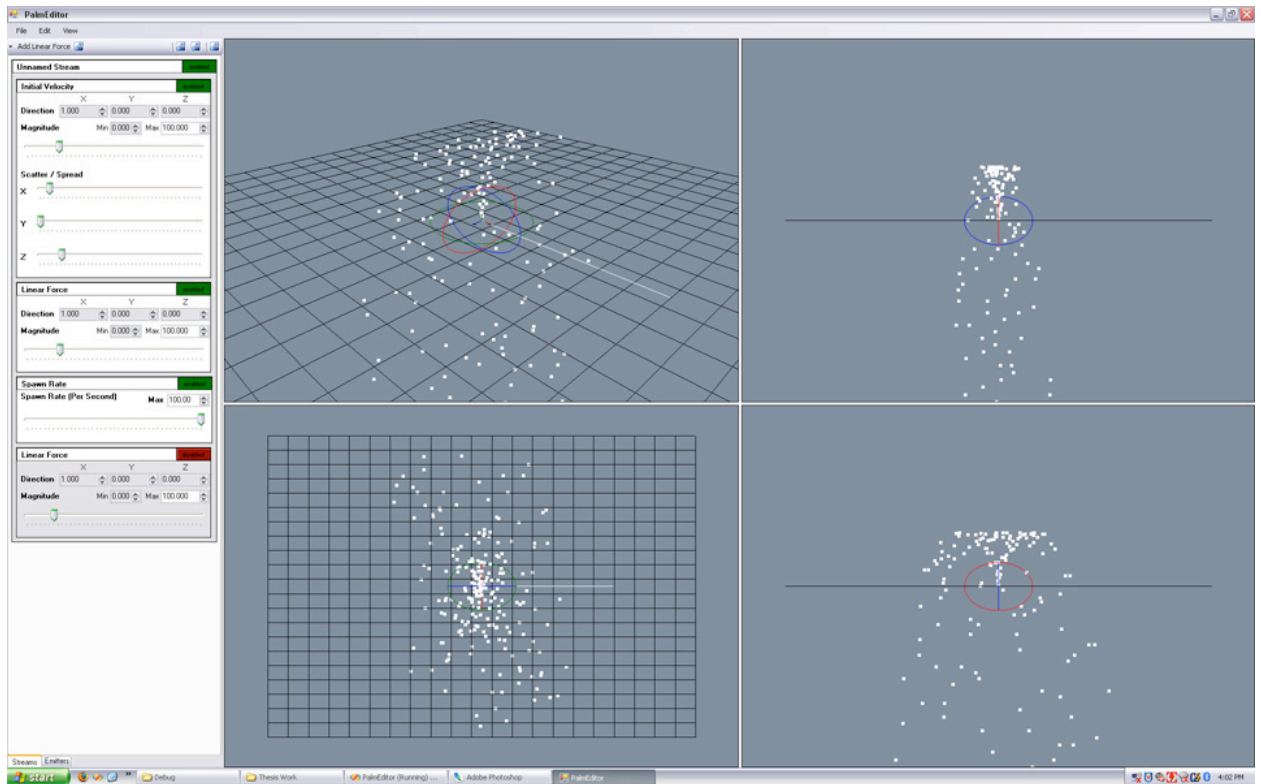


Figure 3.2 – The first iteration user interface. The rotation control shown in the 4 preview panes was not accessible without the emitter first being selected from the list on the left.

The second iteration is what was used for the purpose of this study and provides a clear layout for where tools exist, where to go to add entities to the scene and where to find editable properties for the selected entity. Anything that can be added to the scene is located at the bottom left of the window, with the tools to move, rotate, and scale these entities located above them. Properties are categorized into related groups such as texture, color, size, etc. All properties are found under collapsible tree controls on the right side of the application. The tree control allows more controls to fit in the area than would be possible without a container of some type, and allowing it to collapse allows the artist to view only the controls they are interested in at the moment.

The first system to be built was the new rendering ‘engine’ that would be used to power the tool. C# was chosen as the language to develop the renderer in since that is what the editor is written in and it was desirable to keep the simulation and rendering of particle separate to enforce a particle simulation code base that could be placed into multiple engines. It is also quite possible that the eventual platform for the particle simulation code would be a console, which

has limited support for keyboard and mouse. It is much easier for a designer to produce the particle system on a PC and upload the resulting file to the console to view in game, than to try to make an interface for editing particles on the console itself.

The three tools available to the user for modifying objects in the scene are the rotate, scale, and translate tools. Translate moves the object in the scene. The user can select to move on a single axis by click-dragging on that single axis, or can choose to move along two axes at once by clicking the box that joins the two axes on the translation tool. Translation can be applied to every object in the scene, with exception to child objects which are always moved with their parent. The rotation tool is represented as three circles, each one drawn around the axis about which it rotates. Some objects, like attractors, cannot be rotated because doing so would have no effect on the object. Other objects can be rotated in the tool, but will not exhibit any perceivable change by doing so. One instance of this is rotating a spherical emitter object, since a sphere rotated to any orientation looks the same as it did prior to rotation. Scaling is the final tool available for modifying objects in the scene. Scaling is handled a bit naively for speed reasons. Not all objects can be scaled in all three axes. Spherical emitters can only be scaled uniformly for instance; while box emitters can be scaled independently along each axis. Scaling also only occurs relative to a models local coordinate system, meaning an object that is scaled along the X axis then rotated 40 degrees will look the same as an object rotated 40 degrees than scaled along its X axis. These controls are copies of the same controls from *3d Studio Max*, which should be familiar to most professional game artists.

When a `PalmEntity` is selected all editable properties for the entity are added to the property panel located on the right side of the application window. Individual properties are categorized, and placed in collapsible group containers. This scheme allows for more properties to exist in the property panel than could otherwise be possible without scrolling. The intent is to allow quicker access to the data properties the user wishes to edit. A few properties are editable both on the property page and with a 3D tool. The dimensions of an emitter box would be an example of this. For these properties changes made in either way are reflected in real-time in both controls.

In order to keep all representations of property data in synch with one another the event/delegate model incorporated into C# was put to great use. The object that owns a particular property can attach an update method to the control representing this property data.

Anytime that control is modified the update method takes care of updating all representations of the data. There are potentially three representations of property data while editing, the first being the data in use by the simulation, second the visual representation of the data in the scene, and finally a property control representation of data available in the property list.

The viewport control represents the viewable area and the virtual camera the user is looking through. Camera movement is handled similar to *3D Studio Max* and *Maya*. The user can press and hold the Alt key then click and hold one of three mouse buttons for difference camera controls. Left clicking rotates the camera around the view target, right clicking allow the user to pan left and right as well as forward and backward. The center mouse button allows the user to look around in first person fashion. At any time the user can scroll the mouse wheel to zoom in or out on the look target.

All objects in the editor are represented by a 3D entity in the main window; these are referred to as `PalmEntities`. The user can interact with these objects directly to set positions, orientations, and scales. To accomplish these tasks the user has to first be able to select an object with their mouse. Picking is accomplished by casting a ray from the mouse position on the near plane toward the mouse position on the far plane, and running collision checks against all objects bounding volumes. The bounding volumes defined in the tool are axis aligned bounding boxes (AABB) and bounding spheres. When an object is selected its bounding box is rendered to indicate it is the active object. Any child objects contained within are also only shown when the object itself it selected. These child objects can independently be selected to become the active object.

3.2.2 PalmConnector

The `PalmConnector` class exists to marshal data between the unmanaged C++ particle simulation backend and the managed controls and editor. This class is written in C++/CLI which natively supports mixed mode assemblies and can reference data through proper C++ pointers as well as managed references. Mixed mode assemblies can contain both native machine language instructions as well as Microsoft Intermediate Language (MSIL) instructions. This allows them to be called by .NET components while maintaining compatibility with completely unmanaged components (MSDN). C++/CLI works well for creating mixed assemblies but the task of marshaling data is a development pain. Marshaling data is performed by creating a managed

wrapper class for the unmanaged simulation code. Specifically the PalmConnector is a managed wrapper class for the unmanaged ChiManager class. Every time a new feature was added to the ChiManager, usually to reflect an additional feature of the particle simulation, a wrapper method also had to be written. This is not a complicated procedure but slows down development. Had all of the features of the Chi particle system functionality been designed prior to the development of the tool, this class could have been written all at once.

There is a performance consideration to be aware of when working with a mixed mode (managed and unmanaged) assembly. Any time a managed function calls an unmanaged function and visa versa a special transition sequence called a thunk occurs (MSDN). C++/CLI has special support for thunking which is the fastest available method. Even so the managed to unmanaged barrier should be crossed as little as possible, particularly for methods that need to be called one or more times per frame like when updating the particle vertex buffers. When beginning this project it was not clear how big of a hit this would prove to be, and it was important to establish that it would not prove to be so slow that previewing particle systems would become impossible in real time. It was clear after implementation that while it does not run as fast as a particle rendering system written in pure C++ crossing the managed/unmanaged barrier was not too slow to allow real time preview.

Marshaling data was a necessary evil for two reasons. First it was important that UI programming was made as quick and flexible as possible since it was not clear how many iterations of UI's would need to be created to find something that worked. C# is convenient for Windows UI development because of the included common libraries and ability to design windows (known as forms) with a GUI built into Visual Studio. The second reason that made marshaling data necessary was due to not wanting to program the particle simulation code in C#. Most games are still programmed with C++ to leverage maximum performance from the platform. Building two versions of the simulation code would have been possible but would have slowed development. Slight variations of the implementation details could have also lead to slightly different looking effects between the two simulations. Writing two systems also requires more time, our most valuable resource. For a commercial studio, building two systems essentially increases costs while offering no compelling value (or potentially reducing value if the two systems do not perform precisely the same).

3.2.3 PalmEditor

The PalmEditor project file is the last layer of application division. It contains the entry point of the application and brings up the main window. This project is the glue that binds everything together. All objects in the scene are created and stored within the MainWindow class of this project. MainWindow is also responsible for the creation of the PalmConnector object, creating and initializing the PalmEngine object, and handling input logic for the application.

3.3 Particle Simulation

Although the particle editor itself could run at frame times around 32ms and display a smooth preview to the artist it is important that the simulation code be fast and allow the desired effects to run in the allotted frame time. The slower the simulation code runs, the fewer particles will be available for artists to use. This is so the simulation code can be integrated into the engine and run along with other game code at real time frame rates. To keep the CPU busy and accomplish processing efficiently the particles are divided into streams of data that help minimize data cache misses and allow data processing to be easily split across multiple threads.

3.3.1 Cache Efficiency

The hardware memory caches on today's computers are built to deliver the CPU with as much data as possible, but these caches remain relatively small due to their expense. To achieve efficient cache use it is important to keep your data compact. When data stored in memory is requested and not already available in cache the computer will read it into the cache before sending it to the CPU. This occurs so subsequent accesses will be able to read and write from the cached copy. The amount of memory read into the cache from main memory is not necessarily the same size as the data being requested. All cache reads are performed on cache lines, which are a fixed size and aligned. The size and alignment are platform specific. When the system has to read into memory because data is not in cache (referred to as a cache miss) the data starting at the first alignment address before the data and all data within the cache line will be read into the cache. It is possible to leverage large performance gains by performing all necessary operations on data that's already in the cache. If you can store all necessary data together with no unnecessary data in between you will help minimize cache misses. To improve

cache efficiency the particle systems were designed as a SOA (structure of arrays) as opposed to an AOS (array of structures.) The particle stream class defined in this project represents all the particles of a particular system. Every property that particles can have is kept as a separate array. This provides two primary benefits. First if a particular particle property is missing because it is not needed for this system, the processing step for that property can be completely skipped with a single branch outside of any loop. Second we maintain high cache coherency because all data that will be processed exists in a tight array, with no useless data interspersed. It may be the case that the amount of particle simulation happening on the CPU is not a significant performance penalty, however this structure is easy to implement so it should be considered when designing a new particle back-end.

3.3.2 Stream Processing

Closely related to cache efficiency is the idea of stream processing. As mentioned in section 3.1.1 the particle system classes were designed to operate on entire arrays of data at once. When particle updating is requested each property of the particle system is handled independently when possible. For instance the first property operated on is energy, so all operations related to energy are calculated for every particle in the system before we move on to the next property. Hardware can be designed specifically to take advantage of this style of data processing, and it is in some real world cases. The vector processing capabilities of the Xbox 360 and Playstation 3 are both designed to handle data efficiently in this manner. NVIDIA and ATI both have new graphics cards which are designed around the principle of stream processing. Designing data structures like this has the added benefit of being able to remove some inner loop conditionals to an outer loop. Such would be the case when a particle has optional property data. If all particle data is stored as one structure per particle then for each particle it is necessary to determine if optional data exists and conditionally operate on it. If this data is stored as a separate array than with one conditional the entire array of data can be skipped or processed without inner conditionals. The particle simulation code for this project is setup as a stream process and therefore should perform well on modern consoles.

3.3.3 Particle Motion

The particle system simulation was designed to operate on point masses. Point masses have a scalar mass but negligible volume, they exist at exactly one point (Wikipedia, “Point Mass”). Representing particles in this manner saves considerable processing power when

compared to rigid or soft body particle simulations at the cost of less accurate simulation. Essentially each single point that is simulated represents a volume of particles all which share the same simulation data and therefore results. The loss of accuracy comes from ignoring some of the more difficult to model behaviors of physics, such as thermal expansion, density distributions, angular momentum, or buoyancy. Modeling more advanced physical interactions would be a possibility but it has two major drawbacks. First the simulation needs to occur in real-time so there is a hard limit based on computer processing power. Perhaps more importantly is that particle artists would actually have to give up control over the look of the particle systems to the governing equations. In games we often don't simulate accurately, we simulate in order to make it look the way the designer intends. Since the software being designed for this project was built outside of a game or even a proper game engine there is nothing to collide particles with, so it was decided to leave off any volumetric representation of a particle.

3.3.4 Integration Methods

However even with point masses there remains the question of how accurate a simulation do we want to achieve? There is a performance/accuracy tradeoff to be made when choosing the appropriate integration scheme for particle updates. For particle effects under the influence of strong enough forces it may be necessary to use a more accurate integration model to keep the particles under control. Forward Euler integration is the cheapest and least accurate model available, yet in many instances it provides acceptable performance characteristics. A more expensive but extremely accurate alternative is Rung-Kutta 4 (RK4) numerical method.

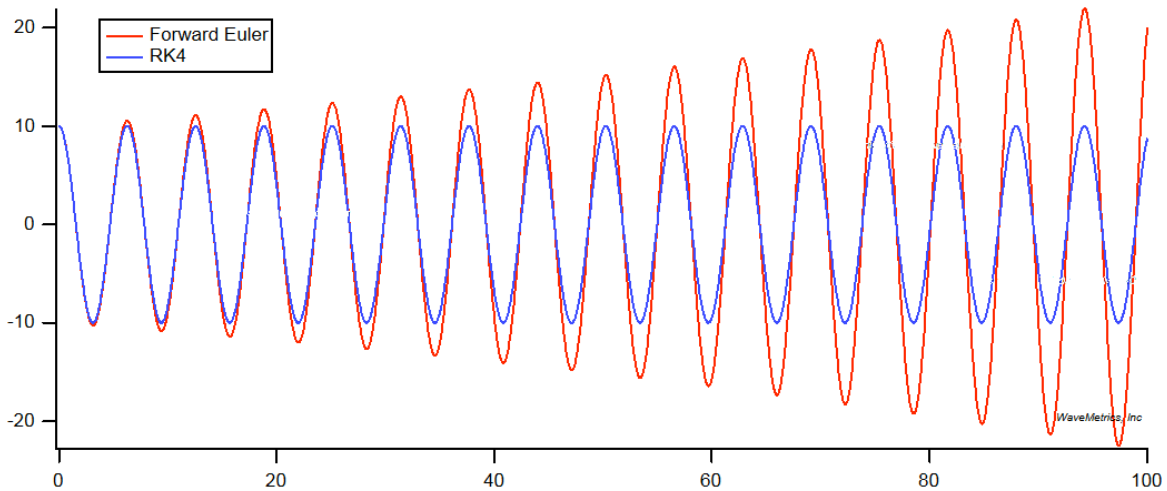


Figure 3.1 Graph of Harmonic Oscillator, simulated with RK4 and forward Euler at 60 fps. Wave amplitude should stay the same size over time. Horizontal axis is time in seconds, vertical is position.

Figure 3.3.2 above shows how the relative inaccuracy of forward Euler integration results in massive error accumulation over time. The diagram shows the position of a harmonic oscillator; it is a waveform that should maintain constant amplitude over time. The blue wave is being integrated using the RK4 numerical method. The red wave is integrated by forward Euler integration. The graph data was obtained by integrating a harmonic oscillator at time steps of precisely $1/60^{\text{th}}$ of a second for 60000 iterations. The diagram shows that after approximately 90[example of a s single individual particle that needs to have a 90 second lifetime? Most particles in games live less than a second, which would show negligible error] seconds the system integrated with forward Euler has almost doubled in amplitude, and it will continue to grow. Forward Euler integration tends to accumulate error over time, which constantly adds energy into the system. This will eventually cause the system to become unstable as additional energy is being added to the system uncontrollably. The particles will increase speed exponentially and explode out of control. Adding viscous dampening to the system is one way to keep the system from exploding out of control, but it has to be tweaked as to not make particle motion seem too slow while still keeping the systems energy under control. Implicit (or reverse) Euler integration would also solve the problem of additional energy being added to the system. For the purposes of the particle simulation forward Euler will be the default and RK4 will be a selectable option for cases when additional simulation stability is desired. More complicated

motions simulated with this system, such as effects including multiple attractors, were shown to become unstable quite easily when using forward Euler integration. This instability made it difficult or impossible for artists to use the control to manipulate effects.

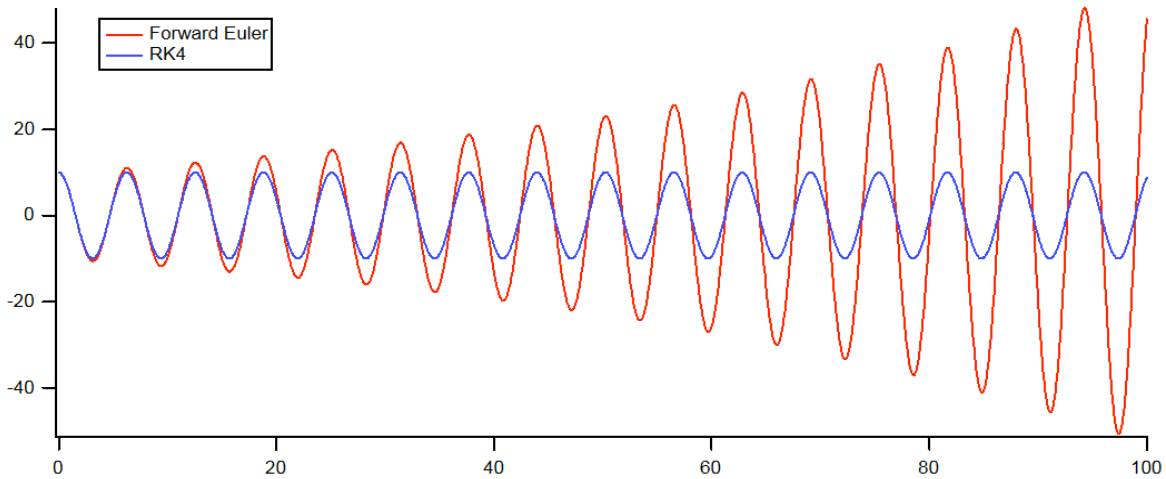


Figure 3.2 The same simulation as above but integrated at 30 fps. Notice the huge error accumulation resulting from calculation inaccuracies associated with forward Euler integration. The waves should maintain amplitude.

RK4 and forward Euler only apply to the updating of a particles position and velocity. There are only two other properties which are stored per particle and that is current and maximum energy. The energy of a particle directly relates to how long it will be alive, specifically when a particle runs out of energy it is removed from the simulation. By storing the current and max energy for a particle it is easy to determine what percent of its energy has been spent. All other particle properties such as color, size, and even animation frame are determined by the percentage of lifetime remaining. This also allows us to store less information about specific particles, and determine more values procedurally.

3.3.5 Forces

All motion in the simulation is calculated from a particles velocity. Velocity is stored per particle and its value is set in two ways. Upon birth every particle is assigned an initial velocity which may be zero. On every update particles may be subject to forces that cause their velocity to change. If we assume every particle has a mass of 1 (unit mass) then a force can be represented as an acceleration, which has a direction and magnitude.

$F = ma$
 $m = 1$ (unit mass)
Therefore $\mathbf{F} = \mathbf{a}$

There are a few ways to specify forces in the application. First is to set one or more gravity entities. Gravity in the application is represented as a single acceleration vector with arbitrary direction and magnitude. More than one gravity entity can be created, which will result in the vector sum of all gravities being used as the systems gravity vector. Gravity is a bit of a misnomer in this case, but its use clearly denoted its affect on a particle.

Attractors and anti-attractors are two more ways for forces to be specified for a particle system. Attractors represent a point mass much like their particle brothers. However attractors do not move under physical properties like particles. Attractors are defined by their world space position and a mass. During particle updates the vector from the particle to the attractor is calculated. Acceleration is calculated and added to the systems gravity to form the final acceleration for the particle by the following equation.

$$\vec{\mathbf{F}} = \sum \frac{M_a (\overrightarrow{\mathbf{P}_a - \mathbf{P}_p})}{|\mathbf{P}_a - \mathbf{P}_p|^2}$$

The resulting affect is that particles accelerate toward the point mass every update. The strength of this acceleration is inversely proportional to the distance between particle and attractor squared. Attractors are easily used to create inward spiral motion in particle systems. Anti-attractors are functionally the same as attractors, but have a negative mass which has the net effect of negating the direction of the acceleration.

screen and therefore overlap more pixels, the expense of rendering pixels many times becomes a very real performance bottleneck. While this issue was taken under consideration no proposed solution to the problem was reached for this project.

3.4 Particle Rendering

Each particle system is drawn as a single list of point sprites. Point sprites are screen aligned squares and require only one vertex to render. They cannot be rotated in a spinning fashion however, that is if the Z axis points directly into or out of the screen the point sprites

would be unable to rotate about the Z axis. The alternative to point sprites is quad billboards, which in general are quads aligned to an arbitrary axis (often screen aligned for particles). Quads are able to rotate about the axis pointing into the screen, but also require four vertices to be defined which would quadruple the data size for particle systems. Quads are not supported in this particle editor. Quads would be absolutely necessary for a production tool, since many effects look much more natural with rotation involved. They were cut from this project in order to spend more time researching and polishing the user interface.

The vertex data that is sent to the graphics card for rendering is relatively minimal, the particles position and energy information are all that is required. When a particle is rendered its color and size are determined procedurally as a function of the particles energy. A color and size table is loaded into vertex shader constants. The size of the tables is arbitrarily limited to 40 entries each, but could be scaled up or down based on specific requirements. The amount of energy spent is calculated as a percentage of maximum energy, that percent is multiplied by the number of used entries in the color or size table less one. The result is a floating point number 'i' between 0.0 and (TableSize - 1). The color or size value at **i** and **i + 1** are linearly interpolated using the fractional part of **i**. This results in a smooth transition in size and color for all particles throughout their life. The size property is modulated based on the eye space depth of the particle, which can be determined from the homogeneous w component of the vectors position after transformation.

3.4.1 Fill Rate

Perhaps the single largest performance bottleneck for particle systems is the fill rate of the device rendering the particles. Particle effects are commonly rendered with alpha blending enabled. This blending is partly responsible for the volumetric appearance particle systems display when rendered, and is considered of the utmost importance. When rendering particle systems, a single pixel will have to be rendered as many times as there are particles overlapping it. When particle systems are very dense, or more importantly when they are very close to the

CHAPTER 4

RESULTS AND ANALYSIS

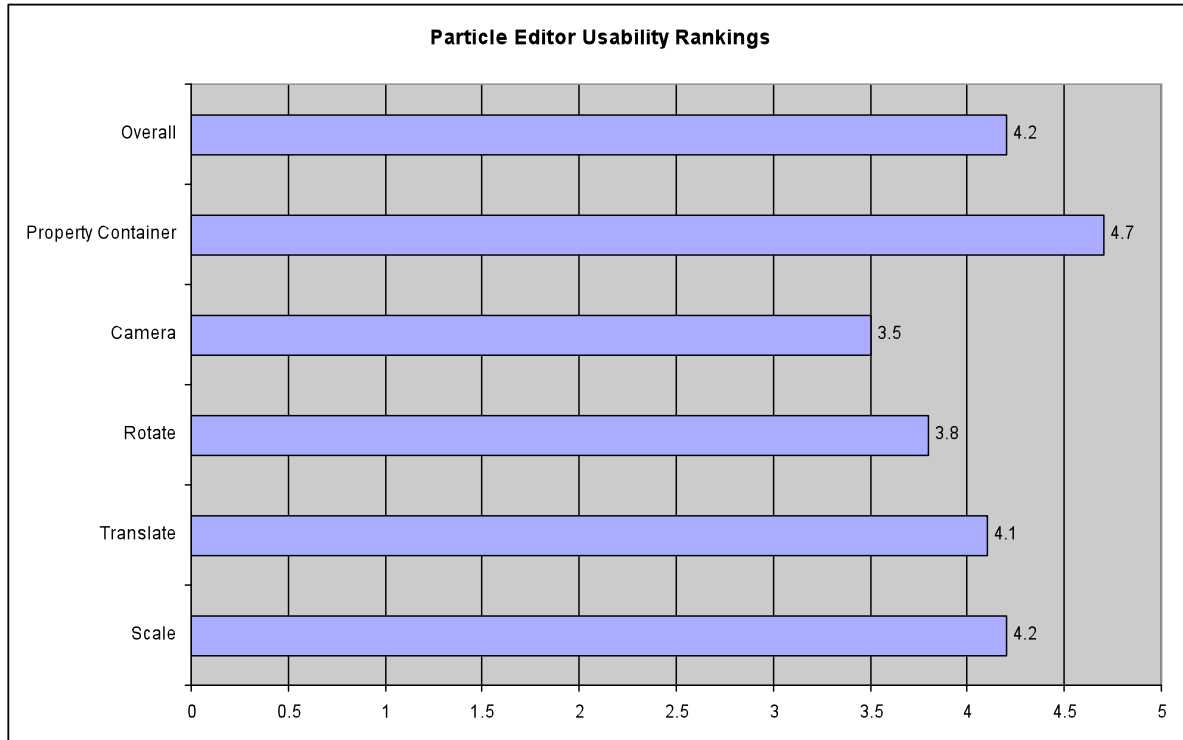
4.1 The Test

In order to determine the usability of specific editor control features and of the editor overall a series of usability tests was conducted. 4 artists, 4 level designers and 2 programmers from the Guildhall at SMU were asked to work with the editor for a while, long enough for them to try out every feature and control and try to design a particle effect of their choice. Before any user began working a short demonstration was given to instruct them on how to position their camera and on basic operation of controls. After the user had tried modeling particle systems for a while they were given a short questionnaire which asked them to rank the major controls usability from 1 to 5, 1 being poor and 5 being excellent. The questionnaire also asked them their preference on manual property entry vs. using the visual controls. Finally they were asked to list any other particle editing tool they had used, and like any features they felt were missing from the editor.

4.2 Results of Usability Rankings

The editor scored an average overall rating of 4.2 out of 5.0 from the test users. According to the sample the most difficult controls to use are the camera controls. This became immediately obvious while observing users navigating the scene. Many time users would find themselves unable to locate the grid on the XZ plane which was put in place to act as a reference for the 3D environment. Users seemed to be a little confused on which type of movement they were performing and how it affected their orientation or position. One area of particular confusion was the difference between panning and zooming forward and backward. Zooming gets you closer to the look target but never past it but panning translates the camera forward. Central to this confusion and perhaps to the camera rotation usability issues is the fact that the look target has no scene representation. The look target is a virtual point in the scene, which confused users. Some of the users communicated that they wanted the currently selected object to act as the point of rotation, and the mouse cursor to act as the zoom target. Changing the

camera controls to act in this way would likely improve the usability of the camera controls. Another commonly cited camera problem was the inability to move along the Y (up) axis. Users thought it would be better to allow them to hold down the left and right mouse buttons to perform this action.

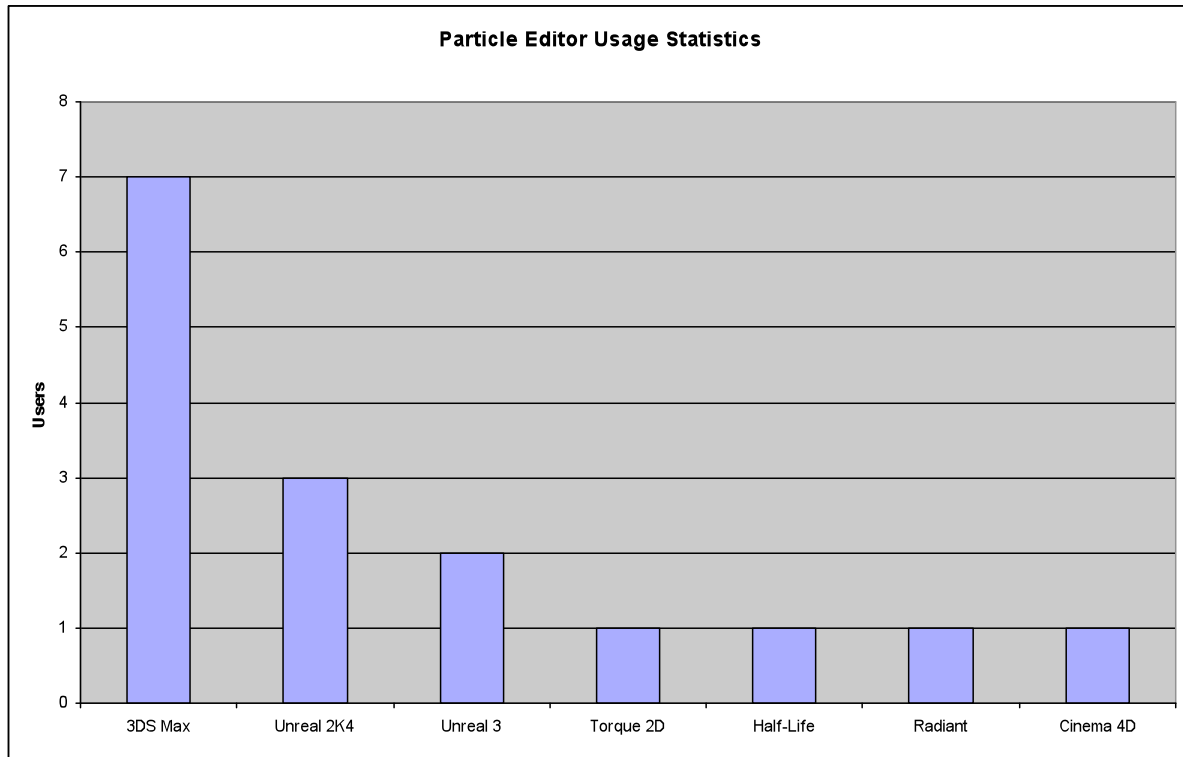


The scale and translate controls scored close to one another with the rotate control being a little less usable. Many users simply thought the controls were too small when viewing them from a distance. Other users were having problems clicking on the different parts of the control. There is no feedback on which part of the control the user is hovering over with the mouse, so users sometimes had to guess which action would be taken if they attempted to use the control. Input sensitivity was another issue cited by users, particularly the users who decided to work in much larger scenes than the editor was intended to work at. There are two particular ways in which the scale, translate, and rotate controls could be vastly improved. First the controls should always appear the exact same size, regardless of the distance to the camera. Second the input to the controls should be based off of cursor location rather than mouse movement. In the case of the translate control this would mean the object would move to where the cursor is on the screen,

so the actual distance moved would be based on how close the camera is to the object. Scaling and rotating controls would also benefit from this type of automatic sensitivity modulation.

The property container control proved to be the highest rated control in use by the editor. This type of control is common in many different applications and users all seem to know exactly how it works, and like to be able to hide information they don't need to see at the moment. Some of the users reported that they wished the property container list was scrollable similar to how *3d Studio Max* allows users to scroll through the large lists in its sidebar.

One question on the survey asked users whether they preferred manual entry of property data or using the visual tools to accomplish the same task. Optionally they could choose both as an answer. Every single user surveyed selected both as their answer, this seems to indicate that people liked having options and each control was usable for different design practices. Some users had a clear idea on the precise value they wanted to use, and in those cases often opted to type in values versus adjusting parameters via the mouse. There was a particular behavior exhibited by most of the users when trying to edit data through the manual controls in the property lists. Many values located there are numeric, and use a numeric up down control that allows the user to type in a value, or increment and decrement the value by a predefined increment value. Almost every user tried to click-drag the up down button on the control. Questioning this behavior led to the discovery that *3d Studio Max* allows users to very quickly increment and decrement the values in a numeric control by clicking the up down arrows and moving the mouse vertically. Based on the amount of users who tried to do this while using the editor it became clear that this feature should be added and is the preferred method for adjusting numeric values.



The final survey question asked the user to list any other particle editing tools they have worked with. This question seeks to determine the types of application the editor was being compared against. By far the most common particle editing application was *3D Studio Max*. By comparison *3D Studio Max* has a much larger feature list, however users reported that max was more difficult to use than the editor built in this project. User feedback suggests that building a tool to meet the specific needs of developers of real time particle effects helps particle designers produce effects quicker than a tool designed to render particles offline.

Another aspect of the user test session and part of the survey was to determine precisely what particle designers wanted to be able to do, and how they wanted to model these systems. The most commonly requested feature was to have more control over color, and size over the lifetime of the particle. The editor allows only for a sequential interpolation between color and size over time. For instance if there are 3 colors in the color table the particle will all interpolate linearly between the first two colors during the first half of the particle life, and the second two colors during the second half. Users want to easily be able to set how long the particle would stay one color, and how long a time the transition from one value to the other would take. One

way to give the user this kind of control would be to allow the user to add colors anywhere they desired on a timeline then fit a curve to the selected values. If different types of curve fits were made available and were complete with control points when applicable, the user would gain a huge amount of control over these properties. Aside from additional control over color and size over time, some users reported their desire for particle to emit at random sizes in a range or random colors from a potential set.

4.3 Observed Usage Issues

The survey is a good tool for determining the usability of some specific editor features but many usability issues became apparent by observing new users trying to use the editor. This section will address the most commonly seen issues.

Users were dissatisfied with the way that all entities added to the scene appeared at the origin by default. Most users wished they could click the add emitter button for instance then click around the screen to add emitters at the locations they clicked. This would help with one issue that came up where a user would add an emitter, not be able to see it because another emitter overlapped it then continue to click add emitter because they believed nothing was happening. This polluted the scene with emitters that the user was completely unaware of. This issue is closely related to another feature that was mentioned as being desirable which is a complete list of all entities in the scene that the user could use to quickly locate and select entities.

When users were trying out the translate, scale, and rotate controls they could be looking at the scene from almost any reference frame depending on where their camera was placed. These controls do not adjust input values based on orientation, so when using the control from the back side the input feels exactly backwards of what would be considered correct.

The users who had *3d Studio Max* experience almost all tried to use hotkeys to select different camera angles, which is not supported currently in the editor. This occurred during many tests suggesting that it is important for any production tool to allow users fast access to common features via hotkeys. This feature would improve user experience without sacrificing anything and would be relatively easy to include were it designed in from the beginning. Due to technical issues relating to keyboard input focus it was not added to this project.

4.4 Results

Tool programmers can learn from this study, or by conducting a similar study of their own, a great deal about how users will approach a tool they haven't used before. It was common to observe users trying to use the controls in a way they were not designed, but made perfectly valid sense to the user. Specifically it is important that if you are going to base the interface of your application on a similar application or one your users are likely to be familiar with you need to go all the way. When a user sees a control that looks the same as one they already know how to use, they will expect it to work in precisely the same way and will come away discouraged if it does not. You can expect that your application will always be compared against a similar product, even if the applications are built for different purposes like real time vs. offline.

Another important take away is that it is better to have fewer polished features and a great interface than many features that are broken or only partly working. When possible don't even let a user know that a soon to be implemented feature is in the works. If it's available or visible to the user it should work completely.

The major focus of this study was to see if the interface features included in this application fulfilled the needs of particle designers to effectively and efficiently create real time particle effects for games. From the feedback provided it seems that the interface is good but has room for improvement. The tools provided seemed appropriate to most users but there were some usability quirks that kept them from being completely comfortable to use. Users reported that the available tools for determining particle motion were good, but they wanted to see more variety.

4.5 Further Work

The design of this application occurred in a sort of vacuum; there was no solid frame of reference for what a designer of real time particle effects would want or need to produce visually stunning effects. In the time after the design much has become clear as to how designers think about particle effects. What seems to be the largest flaw of this implementation is that there is no included concept of a timeline. With this system particles and emitters simply exist and are always emitting particles. Advanced effects require different types of particles to combine together on a timeline in order to produce the desired composite effect. Adding timeline controls to assign periods of spawning to individual emitters would likely provide improved

flexibility[that is certainly very interesting. It is a good example of how the mind-model of a system can be very different for an artist vs a programmer (animation over time vs a continuous system)].

Another flaw is the lack of control of how a parameter changes over time. The included implementation allows for size and color changes over time, but the progression is always linear and evenly distributed across a particles lifetime. Allowing color, size, spawn rates, rotations, almost all parameters to be fit to a curve or interpolate between key frames would also provide enhanced control over effects. Farther work for this project would also include adding or changing functionality of controls to match *3d Studio Max* precisely as well as including hotkeys for common tasks, both suggested by artist feedback.

CHAPTER 5

CONCLUSION

Particle effects play an important role in video games by providing visual cues to players as well as enhancing the visual representation of the games virtual world. Creating new effects is most often the job of an artist or level designer. Particle designers need a tool to enable them to effectively iterate particle designs in real time in order to maximize the quality of the end result and minimize development time. This project aims to fulfill this need by providing a real time particle design interface based closely off of familiar modeling packages like *3d Studio Max*. Preexisting tools such as *3d Studio Max* are not always a practical way to create effects for real time games because of performance requirements. *3d Studio Max* can afford expensive calculations for particles being rendered offline. The feature set provided by *3d Studio Max* is also quite broad. It could take a programmer quite a long time to match every feature available to modify particles, and using any unsupported control would result in an effect which could not be reproduced in the engine.

The research conducted herein suggests that if a tool is to be based off of an existing application then the tool needs to strive to replicate all the functionality of the controls it seeks to replicate. Users also like to have multiple methods of editing the simulation parameters of particle systems. Intuitive interfaces require development and usability research and the more often usability tests are conducted with the intended users the better the final product will turn out.

The scope of the problem that this study was able to cover is limited compared to the potential set of features for both particle simulation and editing interfaces. When developing games assets developers want to have as many options made available to them as possible. Additional simulation or rendering features that could prove useful include particle lighting, fitting function curves to particle property data such as color and size, emitters with different spawn types such as spawning all particles at once. A particle timeline for all properties could prove extremely flexible in allowing designers to time how and when to change particle

properties. Entirely different types of particle systems could also prove interesting, such as beam effects or billboards oriented to arbitrary axis.

If the tool were being developed along side a game it would also be important to have a tool to tie particle systems into levels, attach them to models, and allow them to interact and collide with the environment. [A well written document, if a bit short. But you covered everything nicely and came up with some interesting results. Your focus on analyzing the user experience makes it stand out from “Yet Another Particle Editor”™.... Well done]

REFERENCES

1. Kasavin, Greg. 2004. Gamespot Review, Ninja Gaiden. February 26.
www.gamespot.com
2. Mesnard, Terry. 2005. Amazon.com Spotlight Review, God of War. March 28.
www.amazon.com
3. Gamedev.net Forums
www.gamedev.net
4. Van Verth, James M., Essential Mathematics for Games and Interactive Applications
5. Van der Burg, John. 2000. Building an Advanced Particle System. Game Developer. June 23.
6. Wikipedia contributors, "Point mass," *Wikipedia, The Free Encyclopedia*,
http://en.wikipedia.org/w/index.php?title=Point_mass&oldid=114083074 (accessed March 1, 2007).
7. Microsoft Corporation, "Performance Considerations for Interop (C++)", MSDN,
[http://msdn2.microsoft.com/en-us/library/ky8kkddw\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/ky8kkddw(VS.80).aspx) (accessed March 3, 2007)
8. Microsoft Corporation, "Mixed (Native and Managed) Assemblies", MSDN,
[http://msdn2.microsoft.com/en-us/library/x0w2664k\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/x0w2664k(VS.80).aspx) (accessed March 3, 2007)
9. Screendigest, "Outsourcing in Next Generation Games Development: Delivering cost and production efficiency", screendigest.com,
<http://www.screendigest.com/reports/06outnextgames/NSMH-6MQJSB/sample.pdf>