A Robust and Portable Software Development Kit for Multi-Dimensional Graphical Applications

Michael Sullivan
TJHSST Class of 2004

January 2003

0.1 Introduction

Human beings see in two dimensions. Two dimensional data is input by each eye and the brain interprets three dimensional (3D) images through a triangulation process. Over the past 15 years, computer engineers and software designers have developed complex systems to perform the related task of projecting three dimensional environments onto a two dimensional computer monitor. The computational reciprocal of biological triangulation is now called a three dimensional real-time rendering environment, or 3D engine.

The significance of three dimensional computer graphics far surpasses the simple emulation of biological sight, however. Interactive graphics can be used to synthesize abstract entities, such as objects with no inherent geometry (statistical data) and mathematical surface representations in greater than four dimensions through varying color plates (Foley, et al., 1996). In addition to providing a real time simulation of depth in two dimensional space, 3D engines can also provide a realistic simulation of physical interactions and dynamics. Such physics and dynamics engines serve the dual purpose of increasing the realism of a computer depiction of real world events and providing a medium for scientific and academic visualization and research.

Historically, 3D applications have had to utilize expensive proprietary systems to render 3D scenes and handle physics and dynamics, while providing user input, network support, and data storage schemes separately. This long, expensive process has, in the past, prevented all but the largest companies from developing programs on a 3D platform. As a result, many software development kits (SDKs) have arisen in recent years to simplify the three dimensional application building process. Software development kits exist to serve various purposes. Low-level kits serve as an overlay or extension to a graphics library (such as OpenGL or Direct3d) for software developers. Other SDKs are designed for user access at a medium level, where program code is shared between the SDK and user defined code. High-level SDKs are more like scripting (or modding) interfaces, where a user need not create code, but rather controls a fully coded environment through simple scripts, macros, or configuration files.

There are advantages and disadvantages to each type of SDK. Lower level kits tend to require more knowledgeable developers, who are well versed in whatever programming language the kit supports. Development with such kits tends to be slower and more difficult than on higher level systems, but the software developer is given greater control over the end product. Higher level kits tend to simplify and accelerate application development, but at the cost of developer control. As a result, current SDKs tend either to be too complicated for many potential application developers, or too rudimentary and simplified to create powerful, robust programs. The technical demands or limited applicability of currently available SDKs leave many applications that could aid and clarify scientific research difficult to develop. This project aims to develop a robust and portable SDK which is simple enough to be used by educators and students who are not experienced programmers, yet powerful enough to produce multi-dimensional programs that are not limited in scope or function. Elements of the Software Development Kit discussed in this paper were coded in C/C++ on top of the OpenGL Graphics Library (OpenGL, 2002). Currently all features of the SDK are supported under Windows 98, Windows NT/2000, Windows XP, and Linux.

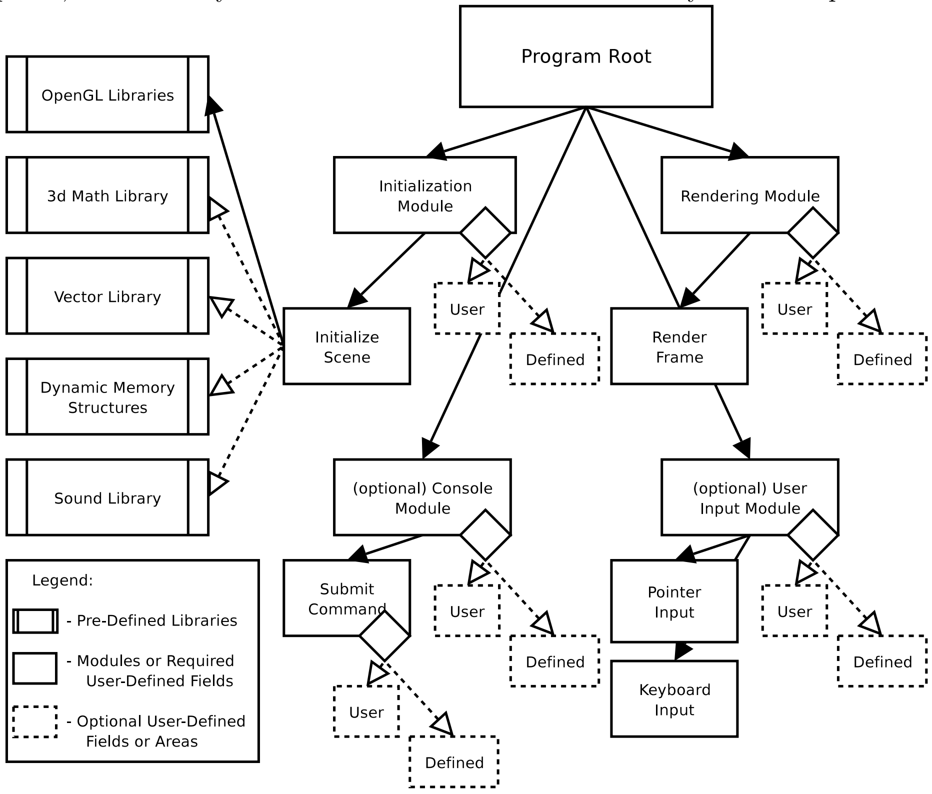
This research report discusses the modular framework of the SDK being developed and presents some of the features that have been coded to date. The aim is not to document all of the kits features but to provide the reader with an overview of the development process and an understanding of how the SDK handles some of the key elements of modeling physical interactions and dynamics in a multi-dimensional environment. The modular framework of the SDK is discussed, along with an overview of the individual modules and libraries that are available to developers. Important engine elements, such as the space partitioning scheme and collision detection method are discussed as well. However, descriptions of many specific features of the built-in modules and libraries have been omitted due to length limitations.

0.2 Modular Architecture

A module-based application is composed of many independent components, each of which carries out a specific function alone or with the help of shared libraries. Modular architecture combines the inherent power of a static, pre-existing core with the scalability of a user-defined program (Gamma, et al.,1995). Standard (built-in) libraries provide tools to handle complex issues (such as texture mapping and collision detection), while allowing the developer impressive flexibility. The modular components of the SDK are as follows.

1. A central root which acts as the engines central nervous system, passing messages to the other modules.
2. A user-defined rendering module which handles the output to the screen every frame.
3. A user-defined initiation module, called at startup, that initializes the variables, libraries, and structures which comprise the application. This module is also called upon shutdown and deconstructs the components of the application in order to avoid memory leaks.
4. An optional user-defined input module which is called every frame to parse and handle input from peripheral devices (keyboard, mouse, joystick, etc.) for the application.
5. An optional user-defined console module which provides direct communication between the user and the application.

Figure 1 presents a schematic of the SDKs modular design. All modules are optional, meaning that if they are not redefined by the developer, the default (built-in) code will be provided by the SDK. However, in the case of the Console and Input Modules, the developer may purposefully omit these functions from appearing in the application being developed and the system will still be fully functional. Some iteration of the Initialization and Rendering modules is required, whether they are the built-in modules or redefined by the developer.



Various standard libraries are included with the core engine, including dynamic data structures (for robust arrays and character strings), mathematic functions for multi-dimensional space, n-dimensional vectors and matrices, as well as optional sound support. These libraries are available to developers (probably those interested in middle level development), though they are not required for software development (only the OpenGL library is required). This approach retains the flexibility and power of a standard (pre-existing) application architecture without restricting the functionality and scope of the end product.

0.2.1 The Standard Modules

If a developer does not explicitly define (or selectively redefine) a module, the SDK has a built-in set of standard modules which should accommodate most applications. In theory, a high-level user could develop an entire application using only the standard modules and libraries (through scripts and configuration files). This ideal solution for those with limited programming skills will become increasingly feasible as additional modules and libraries are developed and contributed to the SDK by the author and other developers. The standard modules include the following features.

1. A Rendering Module which supports an orthogonal matrix (for two dimensional graphs or an overlay to a three dimensional scene) as well as a three dimensional projection matrix. Various time-checks are performed every frame (for time-tempered movement as well as user defined time queues or tables), and the FPS (frames per second) are recorded. Optional debugging features include the running display of screen position (x, y, z) and frames per second.

2. An Initialization Module which is called at startup. The initialization module handles 3D scene (map) files, and passes automatic configuration scripts to the console module. The syntax for the map files is designed to be extremely user friendly, unlike the .bsp files supported by other three dimensional SDKs. Commands are passed to the initialization module by placing a # character before a line. To create and map a geometric figure, Bezier curve, or height map, the only elements needed are a texture name, and the figures placement coordinates. All other elements of a mapped figure are handled by the built-in Texture Mapper (to be discussed later). Comments within a map file follow the ANSI-C standards any code following a double slash (//) or placed between two slash-star tags (/* and */) is not interpreted by the module. An in-depth discussion of the specific commands (code following a #) and the supported geometric shapes (curves and height maps included) is beyond the scope of this paper, due to length constraints.

3. A console module which interprets user commands and scripts. Although this module is optional, meaning that it is not necessary for the creation of an application, it is recommended. The gateway to the console module is a required Submit function, which handles any plain-text submission, and returns either a success (true) or failure (false) to the location which called the console. The built-in console supports many features, such as key mapping, script execution, and system variable manipulation, if allowed. However, an in-depth examination of these specific features is left for another time.

4. 4) A module to handle a users input from peripheral devices. Currently, the built-in user interface module detects (for all supported platforms) the pointing devices (mouse, trackball, joysticks) position, state (mouse clicks, etc.), and can react appropriately to mouse movement. Also, keyboard events (both continuous and instantaneous) are handled. Although the intention is for the user to define the actions taken when various inputs are detected, a set of default actions has been provided. A high-level user can alter the actions taken by the user input module by executing scripts (using the commands bind and unbind).

0.2.2 The Standard Libraries

The backbone of any 3D application is its graphical library. For this reason, OpenGL is automatically included in the SDK. In addition to the OpenGL library (which is required), various other built-in libraries are available to developers. These optional libraries include mathematically intense areas which are commonly used in three dimensional modeling, as well as other classes and structures which are designed to be generally useful when designing a full featured application. The standard libraries include the following features.

1. A 3D math library which contains mathematical elements that are commonly used when dealing with three dimensional space. Such features include support for multi-dimensional matrices, and matrix algebra (determinants and basic mathematical operations).

2. A multi-dimensional vector library which has content tailored to a 3D environment. All vector operations are supported (addition, subtraction) as well as the dot product, cross product, and various controls over the magnitude and direction of a given vector. Other, more specific vector features are supported, such as the determination vectors normal to a surface, and vector transformations due to rotation around an arbitrary axis.

3. Several dynamic memory structures are supported through a built-in library. Templated dynamic arrays are available to provide resizable, reusable storage, and an expanded dynamic string class can ease the formation of a robust and efficient console module. Many specific features exist in both classes, although they will not be covered due to the intended brevity of this paper.

4. An optional sound library is available to developers who choose to add asynchronous or synchronous sound to their applications. Built on top of the FMOD sound library, the built-in sound library provides intuitive and user friendly support for the most common audial needs (FMOD, 2003). The sound library supports 3D sound, and is able to implement many popular types of music compression (mp3, ogg vorbis, wma, and more).

0.3 Portability

One of the most difficult and time consuming aspects of application development is making a program run on a variety of platforms. Due to the number of educational institutions that implement both Microsoft Windows and a version of Linux, support for both of these platforms is imperative for a fully functional product. Also, for compilable utilities (such as an SDK), support for various compilers is necessary, as well.

By detecting certain essential dynamic linked libraries (DLLs) that are common to all versions of Windows, the SDK is able to determine whether or not it is being compiled on a Microsoft owned operating system. If these DLLs are not detected, it is assumed that the parent operating system supports X Window System, the standard graphical system for Linux distributions. After the operating system is detected, preprocessor directives placed within the SDK remove all calls to opposing window systems from the code. This means, using the same code, Windows based compilers will make calls to the Windows Application Program Interface (API), whereas Linux based compilers will use the X Libraries (XLib). By stringently following the ANSI-C standards of programming, as well as obeying certain compiler-specific syntax rules, the SDK has been confirmed as fully compatible on VC++ 6.0, VC++ 7.0, GCC, and Borland compilers.

0.4 The Built-In Texture Mapper

Texture mapping is a technique by which two dimensional images (digital pictures) are mapped onto a three dimensional framework. The OpenGL libraries can handle texture mapping, provided that the user can load the picture into memory and provide information about the orientation and magnification of the projected texture onto the three dimensional object.

Due to the inherent difficulty of this process (loading in the images and figuring out the orientation of the image), a texture mapping class was created to automate such processes for the formats supported by the SDK, as well as an explanation of their unique advantages, are presented in Table 1.

0.5 Mapping the Texture

In order to map a two dimensional texture onto a three dimensional object (whether it be curved or planar), one must first determine the orientation and magnification of the texture. In order to do this, two elements are required for every vertex the two dimensional vertical distance from the bottom of the polygon, and the horizontal distance from the corner (Woo, et al., 1999). Because all three dimensional polygons exist (by construction) in three dimensions, this process entails the development of an algorithm that will convert a 3D polygon into its 2D equivalent, and then compute the horizontal and vertical texture coordinates of each polygon.

Table 1: Supported File Formats

BMP:	Closely tied to the Microsoft Windows API, Microsoft Windows Bitmap (.bmp) files are supported by almost every Windows-based application. The .bmp format supports color depths of up to 32 bits per pixel, and offers a simple byte-wise run-length encoded compression for 4 and 8 bit color depths. High color depth and sheer ubiquity warrant texture mapping support for .bmp images.
JPEG:	Joint Photographic Experts Group (.jpg) files are often used for image transfer and online storage, and have enjoyed relative ubiquity on the internet. Color depth of .jpeg files is up to 24 bit. It is supported by the built-in image mapping module due to its widespread use, versatility, high color depth, and superior compression.
PNG:	A lossless compression of image data (using the Deflate compression method), Portable Network Graphic Format (.png) files can store up to 48-bits per pixel. The .pgn file format was chosen for inclusion due to the current popularity of the format. Also, the .png format is simple to implement and completely portable.
RAW:	The RAW Image Format (.raw) is supported by the image mapping module because of its structural simplicity. Although it may not be the best format for a finished texture (due to its lack of compression), the lossless nature of the .raw format makes it appropriate for testing work before all textures have been finalized.
TGA:	Truevision Graphics Adapter (.tga) files are widely used throughout the computing world. Storing image data with up to 32 bits bits per pixel, .tga files are powerful enough to be used in many paint, graphics, and still-video editing applications. While native (uncompressed) .tga files tend to be fairly large, the file format incorporates a simple run-length encoded scheme which radically reduces file size. The TGA format is widespread and powerful enough to warrant support in texture mapping.

(Murray and VanRyper, 1996)

A simple algorithm developed to automatically map textures relieves a developer of the need to manually calculate this relationship. After triangulating the given polygon and selecting a starting triangle, the texture mapper constructs a line between two of the triangles vertices. The distance between these two points is the horizontal component of the 2D texture. The orthogonal distance to the third point on the triangle can be found using the following equation (Weisstein, 2003a).

$$d = \frac{\left|(V_2 - V_1) \times (V_1 - V_0)\right|}{\left|V_2 - V_1\right|}$$

For any line (V1, V2) and point (V0) where Vn is some point in three-space.(ITALICIZE ME)

After finding the perpendicular distance from the vector, an algorithm involving several algebraic and trigonometric processes is used to find the horizontal component of the texture coordinate for the third point. The texture coordinates for the other vertices are found in a similar way, although for a triangulated polygon, the orientation of previous sections must be taken into account before determining any mapped coordinates.

Another advantage of using a built-in texture mapping class is the simplicity (and cleanliness) of implementing multi-texturing. Multi-texturing is a process that allows more than one texture to be mapped over the same space. Possible applications include variable ground and wall coverings (such as grass, mold, and vines), clouds, logos, etc. The standard texture mapping class has the ability to consolidate all of the various textures that should be mapped on each surface in one concise class.

0.6 Space Partitioning

Three dimensional environments are made up of polygons. Even seemingly curved surfaces are approximated using a polygonal mesh. Unfortunately, when scanning for collisions, there is no native way to determine which polygons could possibly collided with others. Every possibility must be detected, which, for a large environment, can be computationally demanding for even the fastest computer or graphics processor. Two methods of spatial subdivision have been developed for the SDK: spatial occupancy enumeration, and, for increased efficiency, an octree partitioning scheme.(FOOTNOTE 1)

0.6.1 Spatial Occupancy Enumeration

In order to determine whether polygons are near enough to an object to warrant collision scanning, some method of space partitioning is needed. Spatial occupancy enumeration partitions the environment into a number of equally sized units (usually cubes), each of which either holds a part of a polygon, an entire polygon, or is empty (in which case it can be ignored) (Foley, et al., 1996). Once the world is divided thusly, collisions do not need to be checked against a large number of individual polygons. Rather, the set of all proximate cubes can be determined, and only the polygons within these cubes are checked.

0.6.2 Octrees

If a spatial enumeration cuberille has very many cubes (with close to one cube per polygon) or very few (with very many polygons per cube), the space partitioning scheme will not improve the collision detection speed. However, if there are a sufficient number of cube partitions to divide the set of all polygons into smaller adjacent groupings, an appropriate spatial-occupancy array can greatly increase computational efficiency. This is because not all polygons must be checked for collision detection every frame only the ones in proximate cubes. The appropriate number of cubic partitions is difficult to determine since spatial-occupancy enumeration divides the environment uniformly and not all environments have equally distributed polygons. A more efficient variant of a spatial-occupancy cuberille is the octree, which partitions a multi-dimensional space into cubes of non-uniform size, depending on the environmental distribution of polygons (Rogers, 1998).

Whereas spatial-occupancy enumeration divides a 3D space into a pre-defined number of sectors, octrees take into the account the divide-and-conquer power of binary subdivision progressively dividing all full cubes into eight uniform octants (Foley, et al., 1996). Each of these octants are tested for their contents if they are empty, they are ignored. If they contain an appropriate number of polygons, the octant will be a computationally efficient division. Thus, a filled octant (one containing an appropriate number of polygons) will be kept as an end partition. Otherwise, if the octant contains too many polygons to be classified as an efficient space divider, it will be further subdivided (separated into octants), with each subdivision following the same test process as its parent.

Octrees provide an application with the same advantages as a spatial enumeration, with the added advantages of self-determination (for division size) and relative memory efficiency (because of the smaller number of inefficient partitions). Thus, a spatial partitioning scheme utilizing octrees was chosen to replace the relatively inefficient system of spatial occupancy arrays.

0.6.3 Hidden Surface Removal

Although octrees, and even Spatial Occupancy Enumeration arrays can be used for hidden surface removal, the SDK utilizes OpenGLs built-in z-buffer algorithm for depth testing. The z-buffer algorithm keeps track of the closest recorded depth of every pixel projected onto the framebuffer as each polygon is created, and only allows the corresponding element to overwrite the framebuffer if the projected depth is less than that already in the z-buffer (Woo, et al., 1999). Although this approach seems very computationally inefficient, in practical terms, it is the most common and often most efficient hidden surface removal approach used today. This is because special dedicated buffers are often built into modern video cards for depth buffering, and the hardware-based approach of OpenGLs hidden surface removal is, in most cases, much faster and more efficient than any software-based surface removal procedure using space subdivisions.

0.7 Collision Detection

Basic collision detection is a complicated topic in the creation of physics and dynamics engines. In the real world, concrete objects interact with one another, making collision detection redundant. In computer models, however, interactions involve objects without physical substance. For this reason, expansive algorithms must be devised to mimic the properties of real world collisions. The SDK developed in this study utilizes an original collision detection algorithm, designed to be an improvement over existing collision detection methods due to its enhanced robustness, computational efficiency, and simplicity of implementation.

Most iterations of collision detection available to developers today involve the observation of a collision when an object is less than a certain distance away from another object. Distance is measured by the radius of an object relative to a plane via the following generalized algorithm (Weisstein, 2003b).

$$d = \frac{|(V_2 - V_1) \times (V_1 - V_0)|}{|V_2 - V_1|}$$

Two obvious limitations exist with this method of collision detection. First, because it employs the use of radii for determining distance between all non-planar objects, the algorithm only weakly determines collisions between non-convex, or polyhedral, objects (Mirtich, 1998). This means that in terms of collisions, many current physics engines will treat a three dimensional star as a sphere of equal radius. Obviously, this is not a particularly accurate approach. A second problem with many current collision detection algorithms is that they fail to observe collisions in the limiting cases of discontinuous, or sufficiently rapid, movement. In the context of a three dimensional application, such shortcomings may result in a fast moving object completely bypassing a barrier, or another object, which it should(ITALICIZE ME) collide with.

The collision detection algorithm devised for this SDK accounts for both of these limitations. This algorithm is also designed to maximize computational efficiency, which is a serious constraint for all collision detection algorithms. Efficiency is ensured by the full support of a space partitioning scheme for all cases. This acts to limit the number of objects which the collision detection algorithm must examine for any vector of movement.

The collision detection algorithm developed for this SDK is driven by a minimal distance observation principle. The first step of the algorithm involves the location of closest surface points on strictly distinct, non-orthogonal, objects. This allows the algorithm to robustly detect collisions between non-convex shapes (such as stars). Closest points are determined by orthogonal vector decomposition in the direction of projected motion, and by a simplified application of the Gram-Schmidt process of orthogonal projection in the case of interaction between an object and plane (Lay, 2003). Orthogonal decomposition of motion is handled as a usual vector decomposition in three dimensional space.

The second process in the algorithm is the determination of a line in the vector direction of movement. From this, distance projections are made on all objects identified as potential barriers by the space partitioning scheme. Projections are determined by the following generalized algorithm (Weisstein, 2003c).

$$proj(x_o, y_o, z_o) = \begin{cases} x_0 = x_4 + (x_4 - x_5)t \\ y_0 = y_4 + (y_4 - y_5)t \\ z_0 = z_4 + (z_4 - z_5)t \end{cases} \quad : \quad t = \frac{\begin{vmatrix} 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \end{vmatrix}}{\begin{vmatrix} 1 & 1 & 1 & 0 \\ x_1 & x_2 & x_3 & x_5 - x_4 \\ y_1 & y_2 & y_3 & y_5 - y_4 \\ z_1 & z_2 & z_3 & z_5 - z_4 \end{vmatrix}}$$

For any line (V1,V2) and plane (V3,V4,V5) where Vn is some point in three-space, and where (xn,yn,zn) Vn. (ITALICIZE, MATH SYMBOLS)

Due to the nature of orthogonal projection, the distance between the set of points selected from any two objects is guaranteed to be minimal. Further, this closest set of points is not limited to continuous, convex shapes. This allows for robust collision detection between non-convex, or polyhedral, shapes (Mirtich, 1998). The final step in the general collision detection algorithm is a simple comparison of projected (time-tempered) movement against the finite point of collision with a given object. Obviously, if projected movement surpasses an orthogonal projection point on a barrier, the algorithm observes a collision to have occurred, and the amount of actual movement is modified accordingly.

In addition to handling complex shapes, this algorithm correctly identifies collisions in objects moving in a discontinuous, or very rapid, fashion. Because a collision is determined before movement occurs, strict continuity is not required while determining distances. For the same reason, this algorithm accurately detects collisions when objects are moving very rapidly. Thus, the previously noted problem of a fast moving object bypassing a barrier cannot occur under this collision detection algorithm. This is particularly desirable for slower computers, due to the increased relative movement per frame under low frame rate conditions.

0.8 conclusion

The aim of this project was to develop a portable, robust, easy to use software development kit capable of producing programs for a range of multi-dimensional applications. The SDK described here is not meant to be a finished product. Indeed, one of the strengths of an easily used, powerful, open source SDK is that its usefulness continually increases as software developers add new features and optional modules or libraries to the kit(FOOTNOTE 2). Rather than present a final product, this research paper describes the beginning of an evolutionary process.

The guiding principles I followed in building this SDK are that it should be: (1) easy to use, (2) powerful and flexible enough to support a wide range of applications requiring the visualization of physical interactions and dynamics within multi-dimensional space, (3) robust, (4) usable across operating platforms, (5) scaleable, and (6) inexpensive. This SDK differs from existing kits by combining all of these characteristics into one package. Other kits are easy to use, but at the expense of power and flexibility; alternatively, some kits are powerful but require considerable programming expertise. Furthermore, many existing SDKs are oriented towards the creation of a specific class of application, most notably computer games, whereas this project has developed a kit that can accommodate any type of graphical application.

Various design and programming techniques were used to fully achieve the desired results for the SDK. The modular framework upon which the kit is built offers scalability and flexibility to the program, as does the open source nature of the project. As developers create new and different modules and libraries, the power and scope of the SDK will grow accordingly. The default modules and available libraries for the SDK make software creation easy a developer may choose to create or modify as few or as many modules as desired. Software can be developed using this kit without the addition of a single line of code a full application can be created through the use of scripts and configuration files by relying on the default modules. Additional high-level functionality may be possible in the future as new modules are added and the library archive continues to grow. Furthermore, should an experienced programmer require greater control over the application development process, middle and low-level development is possible. Every module and library (with the exception of OpenGL) can be redefined and recreated. All elements (including the OpenGL and FMOD libraries) of the SDK are free for non-commercial use, making the system very inexpensive. Also, built-in operating system detection and cross-platform/cross-compiler code make the SDK easily and naturally portable.

To date, the basic modules, libraries, and features discussed in this paper have been coded and thoroughly tested. All possible submission cases for console and user input modules have been error checked. Also, relative frames per second and startup times (to load an environment) have been logged in order to compare the relative efficiency of various algorithms. The SDKs memory usage was monitored as well. Algorithms that werent robust enough to handle all cases (such as the distance-radius collision detection method) or that failed performance tests (such as the spatial occupancy enumeration partition method) were revised and replaced.

In addition to those elements examined in this paper, many module-specific and graphical elements have been implemented by the SDK to ensure the greatest possible performance from applications. Console features include full keyboard and peripheral device mapping, saving and loading map files, executing scripts, and a developing help library. Map syntax allows for easy multitexturing, texture wrapping, and texture movement/rotation. Automatic rendering support for vertex arrays optimizes code and improves rendering speed without any input from the developer. Additionally, alpha blending and masking are both supported, allowing developers to choose any amount of transparency or opacity. A Lambertian-based lighting scheme supports the dynamic, realistic creation of multiple light sources and the stencil-buffer is used to create basic shadows and reflections. Motion-blur effects blend successive screens and can add fluidity of motion to a graphics. All of these features expand and enrich the abilities of the SDK.

Despite all of the time and effort that has gone into the SDK to date, future implementation of other features will provide increased support for a diverse range of applications. Planned support for existing 3D modeling formats, such as 3D Studio Max (.3ds), Wavefront (.obj), or ASCII Scene Export (.ase) files will allow developers to better build off of the existing work of others. Support for movie file playback (.avi, .mpeg) and the added utility of screen capture commands will expand the abilities of potential developers. Also, development of an advanced particle engine will allow for the dazzling visual effects needed for explosions, dust clouds, and water movement. In addition, integrated netcode will allow many developers to support inter-application communication, something which is often prohibited due to the complicated nature of networked programs. Currently, a prototype of a netcode library is in existence, although cross-platform issues are holding up development.

In this project, I have succeeded in creating a powerful and robust SDK for the development of multi-dimensional applications, as per the guidelines laid out at the beginning of the study. However, like all open source programs, the SDK is an evolving process. Development continues on the built-in functions and features of the kit. Once released, as increasing numbers of new modules and libraries are created, the kit will become progressively broader and more encompassing. Furthermore, with the development of faster and better video and graphics processing capabilities, the field of 3D graphics will continue to change. These advances will surely have an impact on the SDK and the role that it assumes in the future.