Platform independent game engine
by Chad Wesley Armstrong

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University
© Copyright by Chad Wesley Armstrong (2003)

Abstract:
The Platform Independent Game Engine (PIGE) has been designed as a set of tools to easily allow a developer to create a game which can be ported to multiple computing platforms with relative ease. PIGE makes use of cross-platform programming tools such as OpenGL, OpenAL, and the C language.

This gaming engine is separated into two main sections, the core code and the operating system specific header files. The core source code is platform neutral and requires no true modification when compiled and run. However, due to small inconsistencies between platforms, particular areas of PIGE require some functions to be written or modified for a particular operating system. This is where the OS specific header files come in, which are included and removed, depending on which system is running PIGE.

PLATFORM INDEPENDENT GAME ENGINE

by

Chad Wesley Armstrong

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

July 2003

N378
Ar568

APPROVAL

of a thesis submitted by

Chad Wesley Armstrong

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

Ray Babcock	Ray Babcock	7/16/2003
	(Signature)	(Date)

Approved for the Department of Computer Science

Michael Oudshoorn	[illegible]	July 16, 2003.
	(Signature)	(Date)

Approved for the College of Graduate Studies

Bruce R. McLeod	Bruce R. McLeod	7-28-03
	(Signature)	(Date)

STATEMENT OF PERMISSION TO USE

Signature Chad Armstrong

Date 14. July 2003

Dedicated to my
Great Aunt Betty Armstrong

VITA

Chad Wesley Armstrong was born in Longmont, Colorado on November 17, 1977. He is the firstborn to Donald and Dawn Armstrong. He grew up in Great Falls, Montana and lived there until 1996 when he moved to Bozeman, Montana to attend Montana State University. Four years later he received a Bachelor's degree in Computer Science and a Minor in German. He continued on to graduate school at Montana State University where he completed his Master's degree, also in Computer Science.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The Platform Independent Game Engine (PIGE) has been designed as a set of tools to easily allow a developer to create a game which can be ported to multiple computing platforms with relative ease. PIGE makes use of cross-platform programming tools such as OpenGL, OpenAL, and the C language.

This gaming engine is separated into two main sections, the core code and the operating system specific header files. The core source code is platform neutral and requires no true modification when compiled and run. However, due to small inconsistencies between platforms, particular areas of PIGE require some functions to be written or modified for a particular operating system. This is where the OS specific header files come in, which are included and removed, depending on which system is running PIGE.

CHAPTER 1

INTRODUCTION

The computing world of the late 20th and early 21st centuries have been dominated by Microsoft's Windows operating systems (95, 98, NT, 2000, XP), taking control of around 90% of the consumer marketplace. Alternative operating systems such as Linux or Mac OS often have similarly matched software offerings which Windows holds, whether it be web browsers, e-mail clients, or productivity programs. However, there is one area where the Windows-based realm reigns – games. Generally, only the most successful games get ported to Linux or Mac. Even then, there is no guarantee of this happening. When the top games cost millions of dollars to produce, there is little incentive to spend further money and time on minority platforms.

Much of the difficulty of porting games from Windows to other systems is caused by developers using Microsoft-branded tools such as DirectX or the Windows Application Programming Interface (API). When porting, a developer essentially has to translate the game from one language to another operating system's APIs. PIGE is intended to greatly reduce the struggle involved when porting by removing as many system-dependent APIs as possible and relying on open technologies such as OpenGL (Open Graphics Library), OpenAL (Open Audio Library), and the C language, one of the most popular programming languages of the past 30 years.

PIGE follows after the Java methodology: write once, run anywhere. The intended purpose of PIGE is to provide the necessary platform dependent tools for various operating systems (Windows, Linux, Mac), but to allow the core code to remain virtually untouched, so the only modification when compiling for another platform is to change a library name or file header in the main source file. It will not be true platform independence, but the game engine is supported on the top three current operating systems, which are Mac OS, Linux, and Windows. Ideally, all that would be necessary to set up a program using PIGE code would be to add the platform specific header in the main file. Table 1 demonstrates the naming convention used for the header libraries to link to a program's main function.

Operating System	Header
Linux	<pige_linux.h>
Macintosh	<pige_mac.h>
Windows	<pige_windows.h>

Table 1. Header examples.

The 1980s saw computer gaming become a serious form of business and entertainment. Companies such as Sierra On-line and LucasArts created quite a few games in the late 1980s, many with a similar look and feel. This is because they modeled many of their games from the same game engines, which allowed the game developers to work more on the game design, and not on the supporting technology. Like these 1980s game engines, PIGE strives to follow the often touted practice of code reuse.

Since PIGE is available in the public domain, its contents are freely available to anyone with no restrictions. PIGE's free availability is a benefit to the small-scale game developer, or for a person who is interested in learning more about game programming.

Computer	Specifications
Apple Power Mac G4	400 MHz G4 2 Hard drives 20 GB running Mac OS 9 15 GB running Mac OS 10.2 896 MB RAM 16 MB ATI Rage 128 Pro Project Builder
Apple iBook	500 MHz G3 10 GB hard drive, two partitions, running Mac OS 9 and Mac OS 10.1 320 MB RAM 8 MB ATI Rage Mobility Project Builder
PC	1 GHz Intel Pentium III 2 hard drives 3 GB running Windows 98 60 GB running RedHat Linux 8.0 320 MB RAM 32 MB ATI Rage Pro CodeWarrior and gcc

Table 2. Test computer systems.

PIGE was developed on three different machines, two Macintosh computers, and a generic PC running both Windows 98 and RedHat Linux 8.0. Table 2 lists the further specifications of these machines.

This paper is divided into several parts: graphics, sound, porting, and integration. Chapter 2 will discuss more complicated elements of graphics such as collision detection, picking and selection, and loading textures using OpenGL. Chapter 3 will go over the fundamentals of OpenAL and integrating it with OpenGL. Chapter 4 will review the difficulties involved when porting these open technologies across several platforms. Integrating PIGE's graphics and audio components together will be explained in Chapter 5. The appendices hold example code which was used in creating PIGE.

Related Work

If an open source, platform independent, game engine already existed, there would be little purpose for the creation of PIGE. There are quite a few web sites dedicated to the creation of games, such as NeHe [10], Gametutorials.com, idevgames.com, Gamasutra.com, and several other sites committed to gaming technologies for OpenGL and OpenAL.

The most prominent game developer to support multiple platforms is id Software. Many of id's games, especially those in the Quake series, were released for Windows, Linux, and Macintosh. In 1999, id Software released the source code for Quake under the GNU General Public License [5], allowing people to download, view, and use the source code for free as long as they adhered to the GPL guidelines. Two years later, the source code for Quake II was released. Their latest release in the Quake series, Quake III, has not been released under the GPL, but is licensed for $250,000 against a 5% royalty of the wholesale for a single license [8]. The first two Quake games are essentially free under the GPL, but they contain dated technology. Especially in the gaming realm, any technology more than two or three years old becomes recognizably dated. The Quake III engine displays id's most current offerings, but it comes at an expensive price which does not make it a feasible solution to smaller developers.

One of the most inexpensive game engines is the Torque Engine by Garage Games, which was formed from the remnants of Dynamix, after it closed in September 2001. The game engine costs $100 per programmer [6], but it has the restriction that the game needs to be published by Garage Games. The Torque Engine is based on the technology which was used to develop the Tribes games (Tribes, Starsiege, Tribes2). The Torque Engine has been made available for Windows, Linux, and Mac OS 9/X, making it one of the most versatile game engines available.

A trend over the last several years is for companies such as Loki or Westlake Interactive to be solely focused on porting games from Windows to Linux or Macintosh. The disadvantage of this method is that only the most popular games are worth porting, and additional resources of time and money are necessary to fund the port.

Yet, for all of these solutions, none of them have tried to combine OpenGL and OpenAL into an open source game engine, which is the focus of PIGE.

CHAPTER 2

OPENGL

OpenGL was introduced in 1992 as an Application Programming Interface (API) to support the drawing of 2D and 3D graphics [12]. Since then, it has become an industry standard for graphics, extending its capabilities across a wide variety of applications such as medical imaging, virtual reality, mathematical visualization, and computer games. Various technologies have vied with OpenGL for prominence, but none have had its success.

However, OpenGL is somewhat limited in its capabilities, limited mostly to the rendering and drawing of images. GLUT (OpenGL Utility Toolkit) is a window system independent toolkit often used in conjunction with OpenGL to provide extra functionality for windows and I/O communication via the mouse or keyboard. What OpenGL does not take care of, GLUT handles.

OpenGL and GLUT are supported on nearly every popular operating system and computer architecture available. These are optimal choices for PIGE, providing for graphics and user input capabilities.

Documented below are several of the more complicated graphics techniques involved in PIGE. The more rudimentary and basic elements of OpenGL will not be covered in this paper. Refer to either OpenGL Programming Guide [15] or OpenGL : A Primer [1] for an introduction to programming OpenGL.

Collision Detection

With the advent of 3D technologies in the past several years, programmers have made radical changes in how they program applications, especially when regarding computer games. Collision detection is an essential part in 3D games. It ensures that the game physics are relatively realistic, so that an object does not cut through other objects or hovers when it should fall. How well a game can detect collisions is an integral part of the believability and enjoyment of the game. A poorly implemented collision detection system can be a bane to a product, whereas an excellent implementation can produce amazing results.

The two main parts in collision detection are detecting whether or not a collision has happened, and if so, responding to the collision. Discovering if a collision has occurred is the basis of this problem.

While responding to the collision is computationally much easier than discovering a collision, it can still pose several problems in how objects are going to react to each other. In modern computer games, if the character runs into a wall, then the character will either stop or will continue 'sliding' along the wall. However, if this character comes up to a movable box, then the character might start pushing the box instead. Or consider a ball bouncing around in a room. The ball is going to behave quite differently than a person walking around in a room.

The first step is to see if the viewer, or the 'camera', will move through any polygons or planes on its next move. Instead of just calculating if the camera will hit any particular polygon, all polygons are extended indefinitely along their plane. This makes calculations easier to initially perform. If the camera does not intersect the plane, then no calculations are necessary to see if the camera will cross through the polygon itself. This saves some computation by using an easier calculation (with less processing). When a collision with the plane is detected, further processing is done to check if the camera intersects the polygon itself.

2D applications can easily detect collisions by determining if two objects are trying to occupy the same area. If the circle in Figure 1 is trying to get to the triangle, it checks in all available directions. The circle cannot move to the right, since it is blocked by the gray wall, so its only option is to move down. Such a world can easily be represented by a 2D array.

Many older 2D games use a static drawing for the scene, which allows for only a

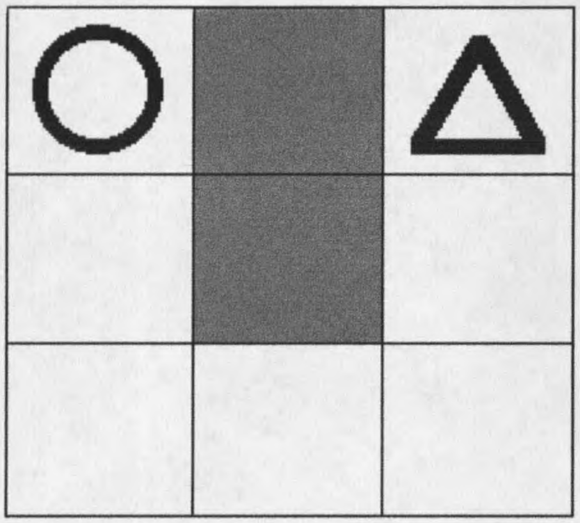

Figure 1. 2D grid.

single perspective, but it simplifies the process of drawing the scene. In comparison, 3D graphics require a large amount of mathematical calculations to render each scene.

Collision Detection Mathematics

A vector is essentially a directed line segment which has direction and magnitude. With graphics, this can be translated as the angle and length or distance. But the magnitude can also represent other factors such as the force or speed of an object. A scalar, as opposed to a vector, only has magnitude, but not direction.

Vectors differ from a point on a Cartesian plane, because the point represents only one spot on the plane, whereas a vector is the difference between two points on a plane. Vectors can be represented in a variety of ways, as shown in equations 1-1 through 1-3. A vector $\underline{\mathbf{V}}$ can be represented as the difference of two points P_1 and P_2 (1-1), the difference of the components of those two points (1-2), or by the constituent vector components (1-3). A 2D world has only x and y components, whereas a 3D world adds the z axis and another component to each vector. Figure 2 gives a visual representation of these equations on a Cartesian plane.

$$\underline{\mathbf{V}} = P_2 - P_1 \tag{1-1}$$

$$\underline{\mathbf{V}} = (x_2 - x1, y_2 - y_1) \tag{1-2}$$

$$\underline{\mathbf{V}} = (V_x, V_y) \tag{1-3}$$

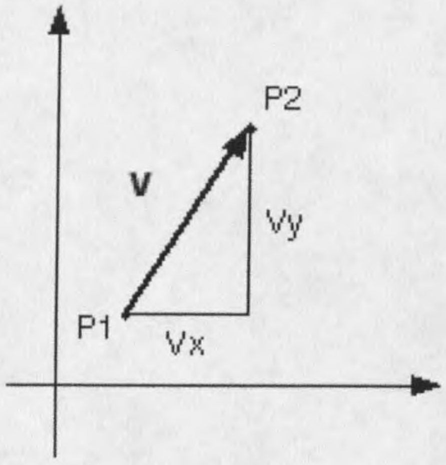

Figure 2. Vector representation.

Vectors play an important role in collision detection by determining how far apart objects are from each other. The magnitude or length of a vector can be found by taking the square root of the sum of the squares of each of the vector's components.

2D Vector: $|\underline{V}| = \text{sqrt}(V_x{}^2 + V_y{}^2)$ (1-4)

3D Vector: $|\underline{V}| = \text{sqrt}(V_x{}^2 + V_y{}^2 + V_Z{}^2)$ (1-5)

To normalize a vector, each component of the vector is divided by the magnitude of the vector. When all of the vector components are added together, they will equal 1. A plane's normal is important to provide realistic lighting and collision detection.

$V_x = V_x/|V|$ (1-6)

$V_y = V_y/|V|$ (1-7)

$V_z = V_z/|V|$ (1-8)

$V_x + V_y + V_z = 1$ (1-9)

Planar Equation: $Ax + By + Cz + D = 0$ (1-10)

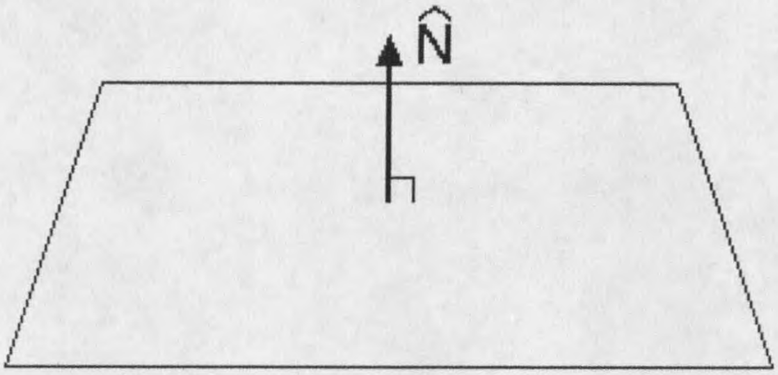

Figure 3. Plane normal.

A plane is defined by three non-collinear points. (x, y, z) from the planar equation are the coordinates of a point on the plane, and the coefficients A, B, C, and D are constants which describe the spatial properties of the plane. A, B, and C can also represent a vector (a, b, c) which is a normal of the plane.

Back-face Culling

Back-face culling is the process of not rendering a polygon if its face is turned away from the viewer. A point can be identified as being on the inside or outside of a plane surface according to the sign of the plane equation. If $Ax + By + Cz + D < 0$, then the point (x, y, z) is on one side of the plane surface. If $Ax + By + Cz + D > 0$, then the point (x, y, z) is on the other side of the plane surface. Otherwise, if the planar equation is equal to zero, the point (x, y, z) is on the plane.

On the average, about half of the polygons in a scene will not be visible by the viewer. Eliminating these unseen polygons can help the performance of viewing the scene since not nearly as many polygons need to be rendered or be involved in the collision detection. This can save an enormous amount of time in calculating where an item may collide. One technique which can save time is the use of Binary Space Partitioning (BSP) trees, which divide up a world into convex hulls to remove unnecessary polygons [9].

To determine whether a plane is facing the viewer or not, the dot product is used to get the angle between the plane's normal and a vector from the viewer's position to a point on the plane. If the angle is between 90 and 270 degrees, then the polygon is facing the viewer. Otherwise, the polygon or plane is not facing the viewer.

Sphere – Plane Collision

One way to detect a collision in a 3D world is the sphere – plane detection method. The demonstration program which was created with PIGE used this technique to detect if the user had bumped into an object. This demo is explained further in chapter 5, and the source code is available in Appendix A. The sphere-plane method is relatively easy to compute since not every polygon of a more complex model has to be compared to the environment to see if a collision has occurred. The viewer or camera can be thought of as one solid entity, such as a ball, instead of a human with several limbs.

Detecting collisions with a sphere tends to be easier to calculate because of the symmetry of the object. The entire surface on a sphere is the same distance from the center, so it is easy to determine whether or not an object has intersected with a sphere. If the distance from the center of the sphere to an object is less than or equal to the sphere's radius, then a collision has occurred.

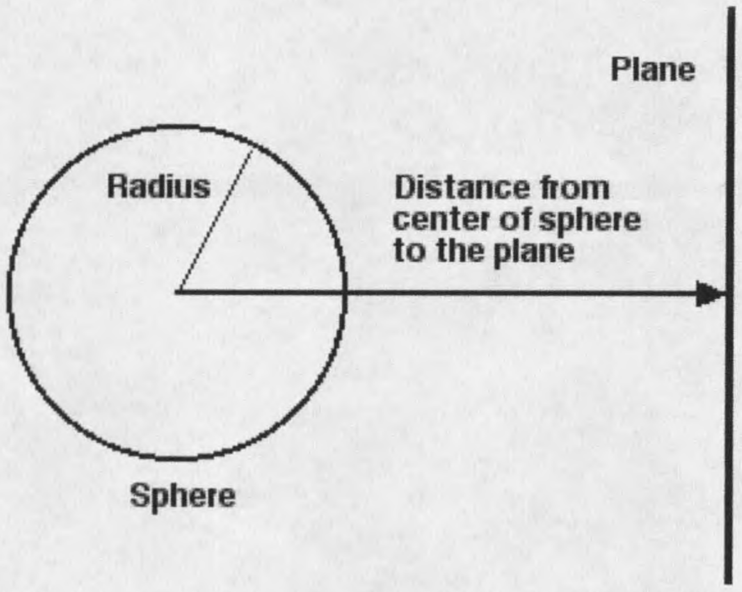

Figure 4. Sphere – plane collision.

The main point is not to let the sphere get too close to the plane. Before doing so, every plane needs to have its own normal vector and D value, which are taken from the planar equation (1-10).

The distance between a vertex, which is the sphere's center point in this case, and the plane is calculated by taking the dot product of the plane's normal and the sphere's position.