

Adam Bates
Ryan Ashford
Dariush Samari
Mike VanDaniker



“1st Person Pac-Man” Formal Proposal

Group Objectives:

- Create a three-dimensional version of the original Pac-man, extending the functionality of the original game.
- Implement various methods discussed in class to gain a greater understanding of the video game industry.
- Successfully complete all major class requirements.

The goal of this project will be to create a unique spin on an arcade classic, implementing the basic rules and goals of “Pac-man” from a 1st or up-close 3rd person perspective. Pac-Man provides a common set of goals and vocabulary with which the group, the teaching staff, and eventual end users are familiar. As a result, the group will be enabled to minimize time in the planning phase and maximize time creating the game. This proposal discusses various phases and objective of development.

Game Outline:

The goal of the original Pac-man game is to collect all of the dots within the maze while avoiding the ghosts that are chasing the player character. Each of 4 ghosts has a unique personality and will pursue the player in a different way. As the difficulty increases, the game becomes increasingly frantic as ghosts are narrowly dodged to complete the dot-collecting quest.

“1st Person Pac-Man” will depict the action of the original game from a first person perspective. The player will no longer have the benefit of a bird’s eye view. This increases the element of maze navigation, as the player and ghosts locations will not be as immediately intuitive. Additionally, it will create a more frantic and exciting game experience.

In order to further extend game functionality and rebalance game difficulty, “1st Person Pac-Man” will include other special modifications, some of which remain unannounced to this point. Elements which have been announced include Pac-Man weaponry, which will be dispersed in the maze like the bouncing fruit in the original. “1st Person Pac-Man” will also include a more interactive interface, and the player will benefit from an in-game informational display. The game will also feature an exciting soundtrack and support user-designed levels.

Phases of Development:

- I. The Original
 - a. This preliminary phase will implement all functionality of the original game.
 - i. Player Character
 1. User-controlled Movement
 2. Dot Collision Detection
 3. Wall Collision Detection
 - ii. Enemy Character
 1. Artificial Intelligence, Chase/Escape Algorithms
 2. 4 unique enemy personalities
 3. Player Character collision detection
 - iii. Maze
 1. Level Design
 2. Ghost Factory
 3. Tunnel Teleportation
 4. Fruit Bonuses
 - iv. Interface
 1. Beginning Menu
 2. Point System
 3. In-game Display
 - v. Miscellaneous
 1. Audio
 2. Victory and Defeat Game States
 3. Camera
 4. Lighting
 - b. This replica of the original Pac-Man will be designed to be easily extendable as the project becomes more sophisticated.
- II. 1st Person Pac-Man
 - a. Implement the Original from a 1st person perspective
 - i. Modified Camera
 1. 1st person view
 2. Close 3rd person view
 - ii. Modified user controls
 1. Mouse controlled movement
 - iii. Modified interface and in-game display
 1. Overhead map display

- III. “Ludicrous Phase”
 - a. Balance game play difficulty, making the game roughly as challenging as the Original.
 - b. Implement Pac-Weaponry
 - c. Additional Possible Extensions
 - i. Alternate game objectives
 - ii. Extend level design system to support additional arcade games

Phase One Assignments:

- I. Adam
 - a. Implement Pac-Man Character class
 - b. Implement Ghost Character classes
 - c. Implement Ghost Factory
- II. Ryan
 - a. Implement Maze classes
 - b. Implement Zone-based object system
- III. Dariush
 - a. Implement level-design system
 - b. Implement game audio
- IV. Mike
 - a. Implement game interface classes
 - b. Implement point system
 - c. Implement game menus

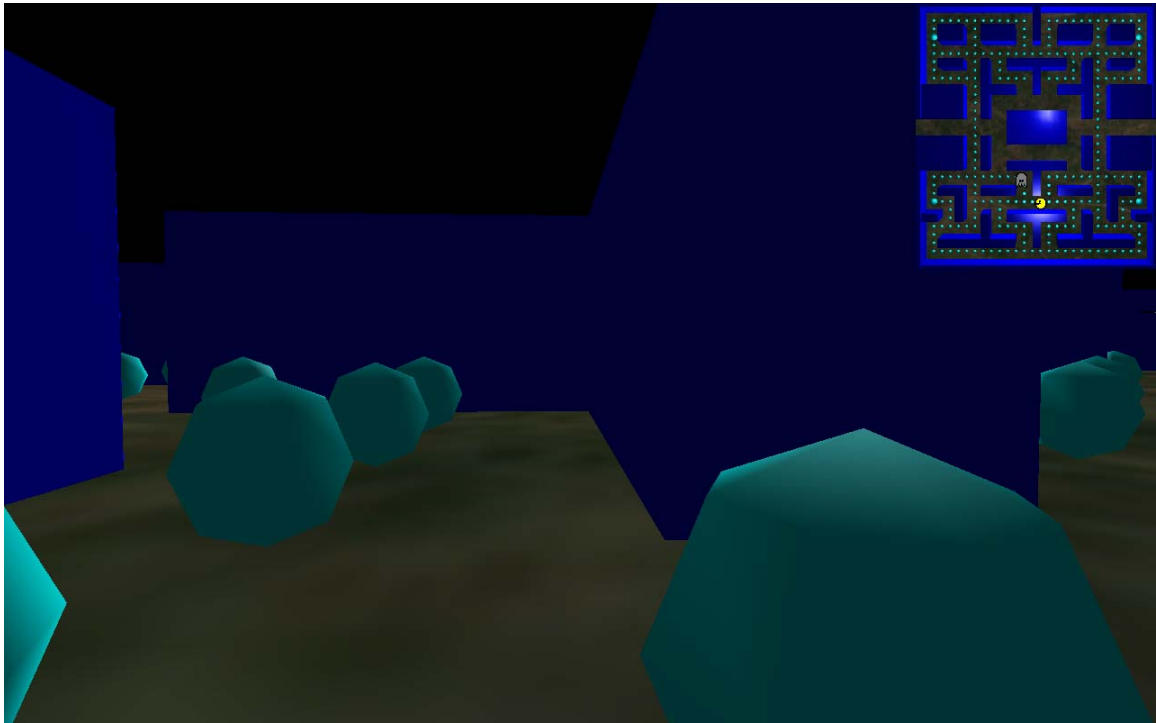
Tools:

The project will be created in Microsoft Visual Studio C++. Additional software is discussed in the appendices.

Sketches & Potential Looks:



Phase One

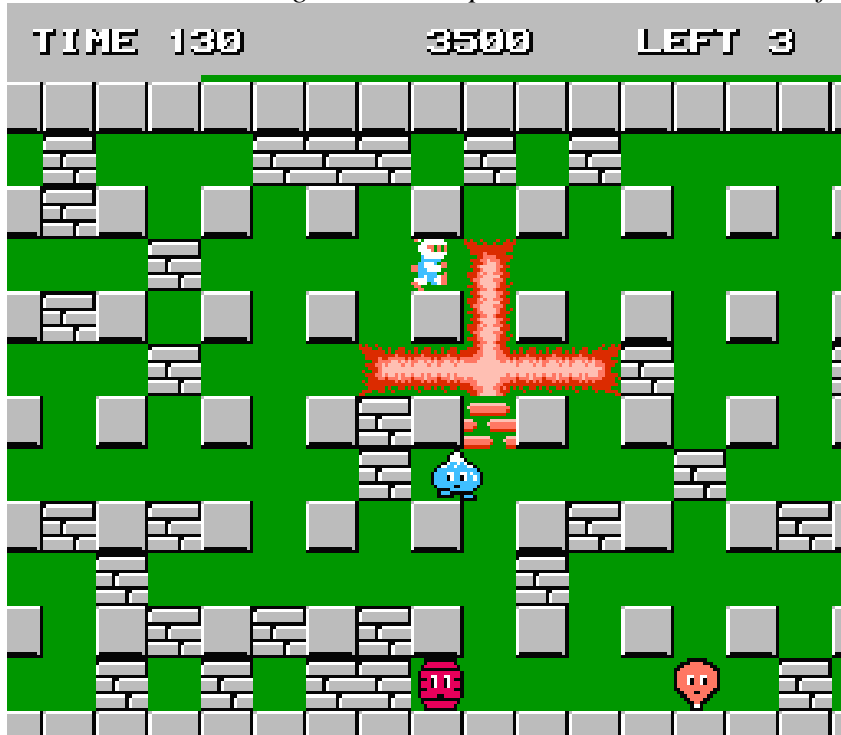


Phase Two



*Phase Three
(Weapon and Interface Graphic taken from DOOM screenshot)*

Examples of Games our Level Designer could implement with minimal modifications:



*Bomberman
(Changes: Destructible Wall Object, Bomb Weapon Object)*



Dig Dug

(Changes: Destructible wall Object, Inflater Object)



Generic and Simple First Person Shooter (DOOM picture)

(Changes: Give ghost weapon capabilities)

Appendices:

The “1st Person Pac-Man” team has already begun to plan a highly detailed implementation strategy for our game. Here are the team-member write ups for various game aspects.

Mike VanDaniker: Interface Explanation

When the program is started up, the player will see the name of the game and options for starting a new game, getting instructions, and seeing the credits. This information will be on a backdrop of one of the levels. If the player doesn't choose one of the options, an animation plays behind the text. One at a time the ghosts each round a corner, coming into view down a central aisle. They approach the camera and line up in the corridor so the player can see them all at once. They hover for a moment as we see their names appear beneath them. After a few seconds they turn blue. They run off the left side of the screen and a moment later we see Pac-Man give chase from the right. Once all of the characters are off screen, the animation will hang for a few seconds and loop from the beginning again.

Ryan Ashford: The Maze System

Description: The maze will support any user design in a given ASCII format. It will be rendered in 3D and will be viewed as if the user were standing down inside the maze instead of looking from overhead. The different features of the maze will be the walls, dots, tunnels, and bonus items. For multiple levels, an array of Maze objects would be stored in the Game class.

Classes Involved:

Maze- Holds all objects pertaining to the physical maze. Composed of a list of zones of a constant size.

Data members:

int sizeX- The number of columns the current level has.

int sizeY- The number of rows the current level has.

static int numDots- The set number of dots in the whole maze.

Member functions:

void render(); - Would render all of the Zone's surrounding a given zone to a certain radius.

Zone- The main object of the Maze implementation. It will hold information such as which walls to be rendered, which zones connect to which other zones, whether there is a tunnel, or if a dot or special item is stored in it. This would be an interface for Tunnel, Item, and Wall.

Data members:

int locX- The X coordinate of the current Zone cell.

int locY- The Y coordinate of the current Zone cell.

Member functions:

virtual void render(); - Would call the render function of whichever subclass object was stored in the current Zone object to be rendered.

Wall- Would store information on which walls to be rendered for a given cell in the game. Values would be read in recursively from the ASCII wall input. Is a subclass of Zone.

Data members:

boolean drawLeft, drawUp, drawRight, drawdown-

Each variable above will let the render function know which sides walls need to be drawn on.

Zone *leftZone, upZone, rightZone, downzone-

Each variable above links to their neighboring Zone. Unsure of the current use of this, but may be useful in the implementation. Could be used to render Zones out to a certain radius.

Member functions:

void render(); - Would render the ceiling and the walls whose boolean values held true.

Tunnel- Stores some unique rendering information related to an idea to use a Stencil buffer to make a seamless traveling experience for the player when traveling through the tunnels. Would allow the player to follow a tunnel to any other tunnel set on the maze(would be preset in the maze design). Is a subclass of Zone.

Data members:

Zone *tunnelLink- Holds the reference to the Zone which this tunnel Zone connects to.

boolean direction- Holds the boolean value of the inward direction(left or right). We'll need to use this so we know which side to use the stencil buffer on as well as which orientation the exit tunnel Zone is.

Member functions:

void render(); - Renders the tunnel as well as the tunnelLink Zone and its visible surrounding Zones. Would also possibly render a physical Tunnel overhang to make the stencil buffer and transition look better. (the stencil buffer idea may not even be needed, but it is something to try)

Item- The Item class would be stored inside the Zone and could be an abstract class to support both Dots and Special Items in a given Zone. This is a

subclass of Zone. This class could be completely done away with and just use Dot and Special on their own as subclasses.

Member functions:

virtual void render(); - Would call the render function of whichever subclass object was stored in the current Item.

Dot- The 'objective' of Pacman. He must destroy all the evil dots with his stomach acid to proceed to following levels. It would be a subclass of Item. This could also be redone to be a DotList between the C intersection locations in the ASCII design.

Data members:

boolean eaten- stores whether the current dot has been eaten or not.

Member functions:

void render(); - renders a sphere at the given position

Special- The class for bonus items. It could support the simple big-pill used to defeat ghosts, bananas and cherries, or even a rocket launcher perhaps. It would be a subclass of Item.

Data members:

int itemType- An int identifier for which type of special item this would be:
0 = Pill
1 = Banana
2 = Cherry
3 = Rocket Launcher
4+ = whatever we want.

Member functions:

void render(); - Would first determine the itemType and then call its given render function.
void renderPill();- Would render a cylinder with two half circles on their ends. Could add a texture if we choose.
void renderBanana();- Not sure how to render this, but a very simple model would not be too difficult.
void renderCherry();- Two spheres with green stalks(cylinders) going up to a knot.
void renderRL();- Extremely simple rendering could have it made of cylinders.

Notes:

The above listing is an extremely generalized format of the layout and will obviously contain many more functions than just render() but I was just going for the basic design so that we could go into detail later.

Also, I'm not exactly sure how the multi-hierarchy of Zone->Item->(Dot, Special) would work, but it's open to discussion. May make sense to just make Dot and Special Zone subclasses and just take Item out of the picture.

We could also add spinning animations to the special objects.

Darius Samari: Level Design And Sound

The sound and music for the game will be configured and programmed using the OpenAL audio library. OpenAL is free and has a fairly good history with current game developers. A few game engines that use OpenAL are the Doom 3 engine, the Unreal engine, and the Torque engine. Many applications have already used OpenAL including Quake 4, Psychonauts, and Lineage 2. According to the OpenAL official website there will no longer be hardware acceleration for DirectSound™ 3D for the Windows Vista. Instead “the OpenAL API will provide a direct path to any native OpenAL driver for delivering premium quality interactive 3D audio for gaming on Windows Vista,” (http://www.openal.org/openal_vista.html). OpenAL is definitely taking a place under the game development spotlight.

There are three main types of objects that are used in OpenGL: the buffers, the sources, and a listener. Buffers are filled with sound data, and each one is attached to a source. Each source is positioned and can be played. The source is heard by the listener, and depending on the position and orientation of the source and listener the sound that the source plays can be heard in different ways. Creating buffers, sources, and listeners, and updating their positions dynamically in a 3D world is necessary in order to present a 3D audio environment. At least one sound device must be opened so that a context can be created and used to store the sources and the listener objects, while the device stores the buffers.

Device #1:

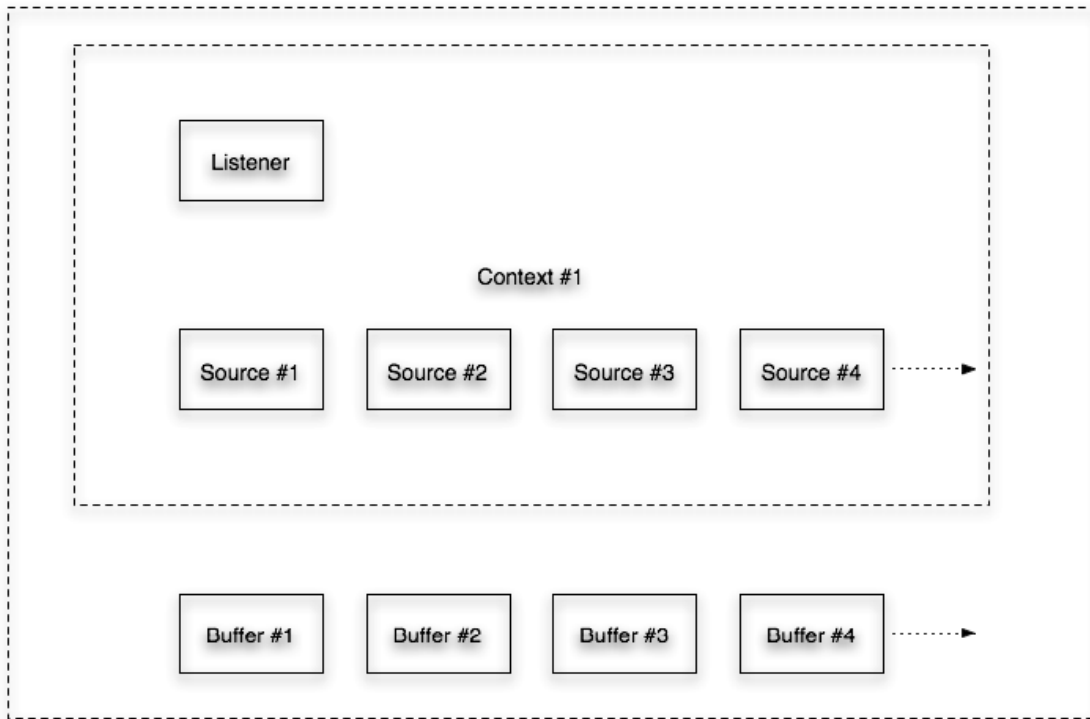


Image was taken from page 7 of the OpenAL Programmers Guide that can be found under the OpenAL website.

By using OpenAL the game will be able to play music and sound effects during game play. The files will probably be of type WAV for good quality and decent loading time. Unlike lighting in OpenGL there can be hundreds of sound source objects throughout the entire 3D world, giving a wide array of sounds and ultimately giving the world a more 3D atmosphere. Several points to consider is having a source object follow around each ghosts and having the listener object simply follow the Pac-Man player. Depending on level design different level music will be played for each level. And perhaps even apply different sounds to each level. For example, the ghosts could make different sounds for each level as they float toward the player. There are many avenues to consider when dealing with sound.

Maze Design

The current program, written by Adam Bates, reads and interprets an ASCII file into a 3D maze. Each character represents one zone of space. The program reads in sets of numbers and interprets them as 3D rectangles, which ultimately become the walls of the maze. Other characters represent empty zones, zones with a dot, a zone with a ghost, and a zone with the player. An example of what the ASCII file for level one looks like is shown below:

30 33 245 100

```

331111111111113311111111111113s
331111111111113311111111111113s
22dddddddddd22dddddddddd2s
22d1111d1111d22d1111d111d2s
22D1111d1111d22d1111d111D2s
22d1111d1111d22d1111d111d2s
22dddddddddd22dddddddddd2s
22d1111d22d11133111d22d1111d2s
22d1111d22d11133111d22d1111d2s
22dddddd22ddd22ddd22dddddd2s
3311111d33111s22s11133d111113s
3311111d33111s22s11133d111113s
3311111d22ssssGsssss22d111113s
3311111d22s1111111s22d111113s
3311111d22s1111111s22d111113s
Tsssssdsss1111111sssdsssssT
3311111d22s1111111s22d1111133
3311111d22s1111111s22d1111133
3311111d22ssssssss22d1111133
3311111d22s11133111s22d1111133
3311111d22s11133111s22d1111133
22dddddd22ddd22ddd22dddddd22
22d1133d1111d22d1111d3311d22
22d1133d1111d22d1111d3311d22
22Ddd22ddddddPdddddd22ddD22
3311d22d22d11133111d22d22d1133
3311d22d22d11133111d22d22d1133
22dddddd22ddd22ddd22dddddd22
22d1111133111d22d111331111d22
22d1111133111d22d111331111d22
22dddddd22ddd22ddd22dddddd22
33111111111111111111111111133
33111111111111111111111111133

```

The three numbers at the top represent the width and height of the maze, the upper limit to the number of possible pellets, and the maximum number of walls allowed.

Each collection of 1s, 2s, and 3s represent a rectangular block of wall. The 'd' characters represent one dot per zone, the 's' characters represent an empty zone, 'G' represents the position of ghost, and 'P' represents the position of the player.

When a number is first read the file reader will keep looking for that same number in the same row, and then increment to the next row until no more of that same number is read. This algorithm works, but can produce poor results when using texture mapping, and can possibly be quite slow when dealing with larger mazes. The texture mapping would be broken up, revealing how the border wall is actually constituted of separate rectangles. Runtime is also wasted as certain facets of rectangles are never shown. A solution would be to have another character, perhaps 'S', signify the start of a new wall. The algorithm would then span the entire wall counter-clockwise along the map until the reader comes back to 'S'. Doing this could possibly take less runtime since there are less 3D shapes to draw, and texture mapping would have a smooth appearance to the pattern, with no breaks.

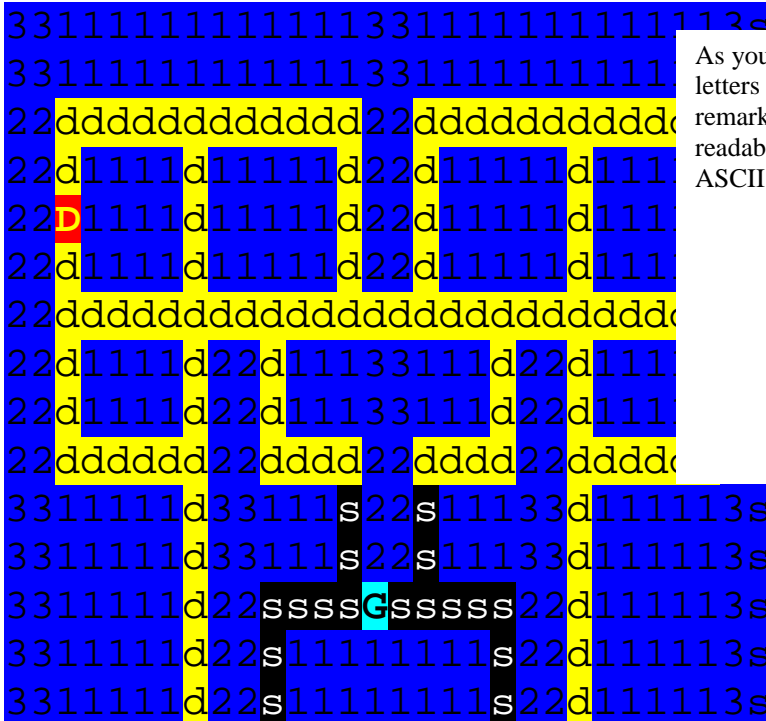
Another important issue is the problem of establishing pellets. The original program reads a single 'd' from the ASCII file and interprets that as one single pellet in that zone. However, the original Pac-Man can have more than one pellet in a single zone. And since the game needs to be as close to the original as possible other techniques of pellet distribution need to be discussed. Ryan proposed an idea of incorporating a 'C' character, which would signify the start of a series of pellets in a single row or column. The group of pellets would have a distancing factor so as to keep more pellets in single row than there are columns. Another idea would be to stick with no 'C' characters, only

'd' characters, but keeping a distancing factor. A single 'd' character would not be a single pellet, but actually the region from which pellets will be established. Two distancing factors would be kept: one for keeping distance between pellets, and another for keeping distance from walls.

Maze Editor

We may want larger mazes which exceed 30 x 32 zones. And visually interpreting large mazes through ASCII characters can be difficult. In order to solve this problem we may need to consider a tool. In this case a maze editing tool would be quite useful when trying to visualize a maze. The tool could read in ASCII map files and create an image that would look like the overhead map in color if the ASCII map file were to be used in the game as a real level. The tool could also be used in reverse! The user could specify a maze size and work with a blank grid. The user could then drag and drop tiles from a toolbox in order to create a maze, and then save it as an ASCII file. Going even further, the tool could be used to create different styles of ASCII files, leading into even more efficient algorithms for maze creation. There is so much you can do with a maze editor.

30 33 245 100



As you can see, having the ASCII letters color-coded can have a remarkable effect on maze readability in contrast to the ASCII map file shown earlier.

