1. Introduction to OpenGL

1.1 OpenGL

Open Graphics Library (OpenGL) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering. Silicon Graphics Inc., (SGI) started developing OpenGL in 1991 and released it in January 1992; applications use it extensively in the fields of computer-aided design (CAD), virtual reality, scientific visualization, information visualization, flight simulation, and video games. OpenGL is managed by the non-profit technology consortium Khronos Group. OpenGL is an evolving API. New versions of the OpenGL specifications are regularly released by the Khronos Group, each of which extends the API to support various new features. The details of each version are decided by consensus between the Group's members, including graphics card manufacturers, operating system designers, and general technology companies such as Mozilla and Google.

The earliest versions of OpenGL were released with a companion library called the OpenGL Utility Library (GLU). It provided simple, useful features which were unlikely to be supported in contemporary hardware, such as tessellating, and generating mipmaps and primitive shapes. The GLU specification was last updated in 1998, and the latest version depends on features which were deprecated with the release of OpenGL 3.1 in 2009.. This interface consists of over 120 distinct commands that we use to specify the objects and operations needed to produce interactive three-dimensional applications.

OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL; instead, we must work through whatever windowing system controls the particular hardware we’re using. Similarly, OpenGL doesn’t provide high-level commands for describing models of three-dimensional objects, however it gives us the ability ro render basic primitives like lines, points and basic 2D Shapes.

In OpenGL, *Rendering*, is the process by which a computer creates images from models. These *models*, or objects, are constructed from geometric primitives - points, lines, and polygons - that are specified by their vertices. The final rendered image consists of pixels drawn on the screen; a pixel is the smallest visible element the display hardware can put on the screen. Information about the pixels (for instance, what color they're supposed to be) is organized in memory into bitplanes. A bitplane is an area of memory that holds one bit of information for every pixel on the screen; the bit might indicate how red a particular pixel is supposed to be, for example. The bitplanes are themselves organized into a *framebuffer*, which holds all the information that the graphics display needs to control the color and intensity of all the pixels on the screen.

1.2 Context and window toolkits

Given that creating an OpenGL context is quite a complex process, and given that it varies between operating systems, automatic OpenGL context creation has become a common feature of several game-development and user-interface libraries, including SDL, Allegro, SFML, FLTK, and Qt. A few libraries have been designed solely to produce an OpenGL-capable window. The first such library was OpenGL Utility Toolkit (GLUT), later superseded by freeglut. GLFW is a newer alternative.

These toolkits are designed to create and manage OpenGL windows, and manage input, but little beyond that.

- GLFW – A cross-platform windowing and keyboard-mouse-joystick handler; is more game-oriented
- freeglut – A cross-platform windowing and keyboard-mouse handler; its API is a superset of the GLUT API, and it is more stable and up to date than GLUT
- OpenGL Utility Toolkit (GLUT) – An old windowing handler, no longer maintained
- Several "multimedia libraries" can create OpenGL windows, in addition to input, sound and other tasks useful for game-like applications
- Allegro 5 – A cross-platform multimedia library with a C API focused on game development
- Simple DirectMedia Layer (SDL) – A cross-platform multimedia library with a C API
- SFML – A cross-platform multimedia library with a C++ API

1.3 GLUT

GLUT (**OpenGL Utility Toolkit)** is a library written by Mark Kilgard; containing utilities for OpenGL programs, which primarily perform system-level I/O with the host operating system. Functions performed include window definition, window control, and monitoring of keyboard and mouse input. Routines for drawing a number of geometric primitives (both in solid and wireframe mode) are also provided, including cubes, spheres and the Utah teapot. GLUT also has some limited support for creating pop-up menus. Some of GLUT's original design decisions made it hard for programmers to perform desired tasks. This led many to create non-canon patches and extensions to GLUT. Some free software or open source reimplementations also include fixes.GLUT provides some routines for the initialization and creating the window (or fullscreen mode). Those functions are called first in a GLUT application:

In the first line always g*lutInit(&argc, argv) is written* . After this, GLUT must be told which display mode is required – single or double buffering, color index mode or RGB and so on. This is done by calling *glutInitDisplayMode()*. The symbolic constants are connected by a logical OR, so you could use *glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE)*.

1.4 SDL

Simple DirectMedia Layer is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D. It is used by video playback software, emulators, and popular games including Valve's award winning catalog and many Humble Bundle games. SDL officially supports Windows, Mac OS X, Linux, iOS, and Android. Support for other platforms may be found in the source code. SDL is written in C, works natively with C++, and there are bindings available for several other languages, including C# and Python. SDL 2.0 is distributed under the zlib license. This license allows you to use SDL freely in any software.

1.5 GLUT vs. SDL

Some of the more notable limitations of the original GLUT library include:
The library requires programmers to call glutMainLoop(), a function which never returns. This makes it hard for programmers to integrate GLUT into a program or library which wishes to have control of its own event loop. A common patch to fix this is to introduce a new function, called glutMainLoopEvent() (FreeGLUT/OpenGLUT), which runs only a single iteration of the GLUT event loop. Another common workaround is to run GLUT's event loop in a separate thread, although this may vary by operating system, and also may introduce synchronization issues or other problems: for example, the Mac OS X GLUT implementation requires that glutMainLoop() be run in the main thread. The fact that glutMainLoop() never returns also means that a GLUT program cannot exit the event loop. FreeGLUT fixes this by introducing a new function, glutLeaveMainLoop(). Since it is no longer maintained (essentially replaced by the open source FreeGLUT) the above design issues are still not resolved in the original GLUT.

FreeGLUT is now very stable and has fewer bugs than the original GLUT. However, there are places where the original GLUT specification did not make clear what order things like callbacks occur and it is possible for application programs that work under GLUT to fail under FreeGLUT because they assume something that GLUT never guaranteed to be true. SDL is a far more widely used API, especially in the games industry, and is regularly maintained and updated. SDL also supports modern OpenGL contexts, which is not supported by GLUT or FreeGLUT. Lastly, in addition to supporting more features it also provides extension libraries for rendering fonts, playing sounds, loading images, and handling input on a variety of devices.

2. PROJECT DESCRIPTION

2.1 Objectives

The objective of taking up this project is to get a first time hands-on experience at developing an interactive graphics application, by utilizing the concepts taught in the in the Computer Graphics class, which is a part of the Bachelors of Engineering in Computer Science and Engineering course in VTU, and to explore a new paradigm in generating maps dynamically in video games, based on a training dataset of maps.

2.2 Software and Hardware Specification

The project was developed on two computers:

- The first computer ran MacOS X 10.11, with a 2.4Ghz Core 2 Duo processor, 256MB NVIDIA GEFORCE 320M and 4GB RAM.
- The second computer ran GNU/Linux Debian Jessie, with a 2.2Ghz Core i3 processor, and 4GB RAM.
- The language and libraries used for graphics were: SDL, SDL-image, SDL-ttf in C++
- The language and libraries used for generation of maps, and ananlyses were: random in Python2.7
- The game is started using a shell script, which first runs the Python script to generate a map, dynamically, and then runs the executable file, which starts the game.

2.3 Description

Fire and Blood is a two-player game in which the terrain maps are dynamically generated, based on a set of maps fed as training data for the game to understand what kind of maps are to be used while the game is played. At the time of the current work, paradigms such as this are not extremely widespread: maps in games are either statically defined, or generated based on probabilistic approaches, which do not use training data, such as bias weights associated with each game object when the game generates a map for the user to play on, etc. The primary advantage of this approach is that rules pertaining to map generation need not be explicitly articulated in code to generate playable maps; instead, the game itself is able to understand the preffered trends in placing different terrain objects at different locations on the map, giving the user or the programmer the ability to generate maps as per their preference, but not in a deterministic way.

The game uses the library SDL, which is the Simple DirectMedia Layer library. SDL supports the latest version of OpenGL, which is version 4, while GLUT supports only upto OpenGL version 2. The viewpoint in the game is a top-down view, and the game involves moving around the sprite of a boy with a gun or that of a zombie. The game is won by reducing the health of the opponent to zero, and this is done by shooting the gun, in the case of the boy, and by coming in direct contact with the boy, in case of the zombie.

The game is fairly easy to learn, and does not change in difficulty. The controls are intuitive, similar to most shooter type vide games: the movement requires the W-A-S-D type control for the boy and arrow keys for the zombie, and for the boy to shoot, the mouse needs to be pointed and clicked. There are no power-ups in the game, and the ammo of the gun does not decrease, either. In a nutshell, the game is simple to play, and does not require much sophistication at the user's end.

3. API’S USED

1. SDL_Init:
    a. int SDL_Init(Uint32 flags)
    b. Use this function to initialize the SDL library. This must be called before using any other SDL function.

2. SDL_SetHint:
    a. SDL_bool SDL_SetHint(const char* name,const char* value)
    b. Use this function to set a hint with normal priority.

3. SDL_CreateWindow:
    a. SDL_Window* SDL_CreateWindow(const char* title, int x,int y, int w, int h, Uint32flags)

   b. Use this function to create a window with the specified position, dimensions, and flags.

4. SDL_CreateRenderer:
   a. SDL_Renderer* SDL_CreateRenderer(SDL_Window* window, int index, Uint32 flags)
   b. Use this function to create a 2D rendering context for a window.

5. IMG_INIT:
   a. int IMG_Init(int *flags*)
   b. Used to initaliase the SDL_Image Library, for a given set of image formats specified as flags. Flags like IMG_INIT_PNG, IMG_INIT_JPG etc. need to be provided as bitwise OR'd set of image formats.

6. TTF_INIT:
   a. Initialise SDL_TTF Library

7. TTF_OPENFONT:
   a. TTF_Font *TTF_OpenFont(const char **file*, int *ptsize*)
   b. Loads a TTF Font file at a given size

8. IMG_QUIT:
   a. Quits the SDL Image Library

9. TTL_QUIT:
   a. Quits the SDL TTL Library

10. SDL_QUIT:
    a. Quits the SDL Library

11. SDL_DestroyRenderer :
    a. Destroys an SDL Render Destination Object

12. SDL_DestroyWindow:

a. Destroys an SDL Window Instance

13. IMG_Load
    a. SDL_Surface *IMG_Load(const char *file)
    b. *Loads an image of specified format from a file*

14. SDL_SetColorKey:
    a. int SDL_SetColorKey(SDL_Surface* surface, int flag, Uint32 key)
    b. Use this function to set the color key (transparent pixel) in a surface.

15. SDL_CreateTextureFromSurface
    a. SDL_Texture* SDL_CreateTextureFromSurface(SDL_Renderer* renderer, SDL_Surface* surface)
    b. Use this function to create a texture from an existing surface.

16. SDL_FreeSurfaces
    a. void SDL_FreeSurface(SDL_Surface* surface)
    b. Use this function to free an RGB surface.

17. SDL_DestroyTexture
    a. void SDL_DestroyTexture(SDL_Texture* texture)
    b. Use this function to destroy the specified texture.

18. SDL_SetTextureColorMod
    a. int SDL_SetTextureColorMod(SDL_Texture* texture, Uint8 r, Uint8 g, Uint8 b)
    b. Use this function to set an additional color value multiplied into render copy operations.

19. SDL_RenderCopyEx
    a. int SDL_RenderCopyEx(SDL_Renderer* renderer, SDL_Texture* texture, const SDL_Rect* srcrect, const SDL_Rect* dstrect, const double angle, const SDL_Point* center, const SDL_RendererFlip flip)
    b. Use this function to copy a portion of the texture to the current rendering tar-

get, optionally rotating it by angle around the given center and also flipping it top-bottom and/or left-right.

4. Implementation Details

The game has been divided into two broad categories, one part is the Python Script that generates the game layouts based on supplied training data. The other part is the C++ implementation of the game and graphics, using the SDL API. The game portion has been divided into their own header file structure as follows:

4.1 Map Generation Approach

Maps in the game are generated by analysing a set of training maps for the probability of occurence of each object in a block. A block is a an area on the map where the image of a corresponding tile may be placed. The layout of the maps in the game are illustrated in Figure 1.

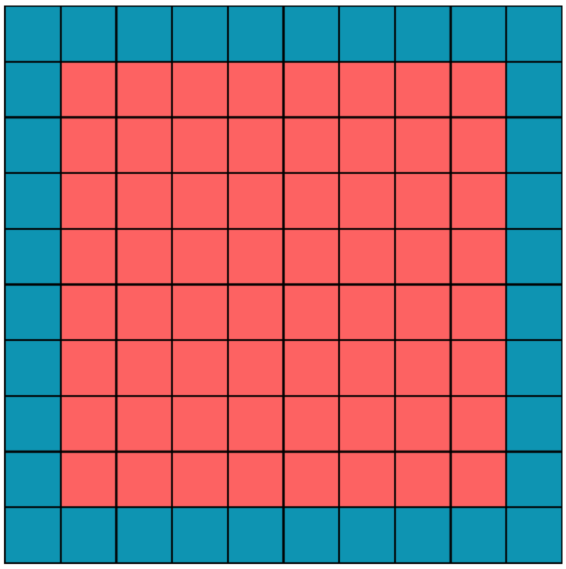


Figure 1: Layout of a map

The red blocks correspond to the regions where the floor tile will be an image of stone or lava: the user may walk on the stone areas, and may not walk on the lava areas. The blue blocks represent a border, which is also lava: hence, the users may not cross over the border and go outside the viewing window of the game. Coming in contact with the lava diminishes the players' health.

In the current approach, the border encompassing the playable area is always constant. The portions of map corresponding to the red blocks may change, based on the training maps: those are the parts where the randomizing aspect of the game is utilized.

Training Maps:

The maps fed to the game are of the same format as explained in the previous section.

The outcome on each block may be a floor tile, or a lava tile. Hence, the outcome is binary. If the number of maps fed to the system is N, stone and the probability that a stone tile occurs at any block x is given by $P(X|\theta_1¿$, then the probability that a lava tile occurs on the same block is given by $P(X|\theta_2¿=1-P(X|\theta_1¿$, where theta1 represents the condition of the occurrence of a stone tile, and theta2 represents the occurrence of the floor tile.

In the current work, we hence work by considering only the probability of occurrence of stone tiles. This is given as:

$$P(X|\theta_1¿=\frac{1}{N}\sum_{1}^{N}1, if\ i(X) is\ a\ stone\ tile$$

Dynamically Generating a Map:

While generating a map, at each block X, a stone tile is placed with the probability P(X | theta1). The map is progressively built up, and a routine checks that every playable tile is accessible by the players, that is, each stone tile is reachable from every other stone tile.

Once the map is generated, the gameplay begins, and each player has to try to win over the other.

4.2 Header Files

- main.cpp – Contains the actual main function. Here we create the Player and projectile objects, and we initialize all the libraries we use, as well as load all game assets. The function is organized as EVENT HANDLING, LOGIC, RENDERING. Finally we call a close function which closes all libraries and frees all pointers.

- functions.h – Contains all functions
- fireball.h – Contains functions and properties of the player projectile
- constants.h = Contains all game constants like health of players, and various enumerators for game and player states
- globals.h – Contains objects and variables used globally throughout the program like the SDL_Window, SDL_Renderer, and Animation Spritesheets.
- texture.h – Texture wrapper class for various texture behaviours
- timer.h – Timer wrapper class which enables easy creation, use and stopping of ingame timers.
- tile.h – Class containing functions and properties of all the dungeon tiles
- players.h – Contains three subclasses. The Main Player class which contains common behaviours between the Zombie and Kid, and then a separate Zombie and Kid Class, which are inherited from this base Player class.

4.3 Classes

```
class Player
{
   public:
      //Returns Angle of Player
      double GetAngle();

      //Returns The Players Collision Rectangle
      SDL_Rect GetRect();

      //Returns and Sets Player State
      playerstates GetState();

      //Sets the Player State to a Particular Value in the State Enumeration
      void SetState(playerstates);

      //Returns X Position of the Player
      int GetXPos();

      //Returns Y Position of the Player
      int GetYPos();

   protected: //Do Inherited Classes Will Inherit as Private
      //Player State Variable
```

```
        playerstates state;

        //The X and Y offsets of the Player
        int posX, posY;

        //The velocity of the Player
        int velX, velY;

        //Angle Kid is Facing
        double angle;

        //Kid's collision box
        SDL_Rect collider;
};

//The Protagonist Behaviour Wrapper Class
class Kid : public Player
{
    public:
        //Initializes the variables
        Kid();

        //Takes key presses and adjusts the Kid's velocity
        void HandleEvent( SDL_Event& e);

        //Moves the Kid and checks collision with appropriate Tiles
        void Move( Tile* tiles[], Uint8*, Uint8*, Uint8*, playerstates state );

        //Displays and Animates Kid On Screen
        void Animate(int frameNum, Uint8 r, Uint8 g, Uint8 b, playerstates state );
};

//The Antagonist Behaviour Wrapper Class
class Zombie : public Player
{
    public:
        //Initializes the variables
        Zombie();

        //Takes key presses and adjusts the Zombie's velocity
        void HandleEvent( SDL_Event& e );

        //Moves the Zombie and checks collision with appropriate Tiles
        void Move( Tile* tiles[], Uint8*, Uint8*, Uint8* );
        //Displays and Animates the Zombie On Screen

        void Animate(int frameNum, playerstates state, Uint8 r, Uint8 g, Uint8 b)
};
class Tile
{
public:
    //Constructor
    Tile(int x, int y, int tileType);

    //Draws The Tile
    void Render();

    //Returns The Tile Collider, for Collision Detection
    SDL_Rect GetRect();

    //Returns the Tile Type
    int GetType();

    //A Texture Index Associated With The Tile
    int texture;

private:
    //Tile Collision Box
    SDL_Rect collider;

    //X and Y Position of the Tile
    int posX, posY;

    //The Type of Tile
    int tileType;

    //RGB Color Values, for Color Modulation
    Uint8 r,g,b;
};//The application time based timer
class LTimer
{
public:
    //Initializes variables
    LTimer();

    //The various clock actions
    void start();
    void stop();
    void pause();
    void unpause();

    //Gets the timer's time
    Uint32 getTicks();

    //Checks the status of the timer
    bool isStarted();
    bool isPaused();

private:
    //The clock time when the timer started
    Uint32 mStartTicks;

    //The ticks stored when the timer was paused
    Uint32 mPausedTicks;

    //The timer status
    bool mPaused;
    bool mStarted;
};//Texture Wrapper Class
class Texture
{
    public:
        //Initializes variables
        Texture();

        //Deallocates memory
        ~Texture();

        //Loads image at specified path
        bool LoadFromFile( std::string path );

    //#ifdef _SDL_TTF_H
        //Creates image from font string
        bool LoadFromRenderedText( std::string textureText, SDL_Color textColor );
    //#endif

        //Deallocates texture
        void Free();

        //Set color modulation
        void SetColor( Uint8 red, Uint8 green, Uint8 blue );
        //Set blending
        void SetBlendMode( SDL_BlendMode blending );

        //Set alpha modulation
        void SetAlpha( Uint8 alpha );

        //Renders texture at given point
        void Render( int x, int y, SDL_Rect* clip = NULL, double angle = 0.0, SDL_Point*
center = NULL, SDL_RendererFlip flip = SDL_FLIP_NONE );

        //Gets image dimensions
        int GetWidth();
        int GetHeight();

    private:
        //The actual hardware texture
        SDL_Texture* mTexture;

        //Image dimensions
        int mWidth;
        int mHeight;
};
class FireBall
{
    public:
        //FireBall Constructor
        FireBall(int, int, double);

        //Animates and Displays the FireBall
        void Animate(int frameNum, projectilestates state);

        //Returns the FireBall Collider
        SDL_Rect GetRect();

        //Moves the FireBall Based On Player Orientation When Fired
        void Move();

    private:
        //The X and Y offsets of the FireBall
        int posX, posY;

        //The velocity of the FireBall, along each axis
        int velX, velY;
```

```
	//Angle FireBall is Facing
	double angle;

	//FireBall's collision box
	SDL_Rect collider;
};//Initializes All Necessary Libraries
bool InitLibraries ( void );

//Loads All Assets
bool LoadMedia( Tile* tiles[]);

//Closes All Libraries and Unneeded Files
void Close( Tile* tiles[]);

//Box collision detector
bool CheckCollision( SDL_Rect a, SDL_Rect b );

//Sets tiles from tile map
bool SetTiles( Tile *tiles[] );

//Checks collision box against set of tiles
bool TouchesWall( SDL_Rect box, Tile* tiles[] );

//Check collision between players
bool PlayerCollision( Kid* kid, Zombie* zombie );

//Finds the angle wrt X Axis; Given Any Two Points
double FindAngle (int x1, int x2, int y1, int y2);

//Sees If The Fireball Hits ANY Object
bool FireBallCollision( Kid* kid, Zombie* zombie, Tile* tiles[] );

//Sees If The Fireball Hits the Zombie
bool ProjectileImpact(Zombie* zombie);
```

5. Source Code

```
##mapgenerationscript.py
import numpy as np
import csv
import os
import random

def randomTileClass(bias):                                  #FUNCTION TO DETER-
    MINE IF A BLOCK SHOULD BE TILED, OR NOT
    r = random.uniform(0, 1)                         #bias IS THE PROBABILITY OF A
    TILE OCCURING IN A BLOCK
    #print r
    if (r <= bias):
         return 0

    else:
         return 1

########################################


def findLeftNeighbour(current_row, current_col):
    global NEW_MAP
    global ROWS_LIMIT
    global COLUMNS_LIMIT
    leftNeighbour = -1

    if (current_col-1) >= 0:
         leftNeighbour = (current_row, current_col-1)

    return leftNeighbour
########################################
MAPS_PATH = './Maps/'                                          #PATH TO MAPS DI-
    RECTORY
MAPS = os.listdir(MAPS_PATH)                        #LIST CONTAINING NAMES
    OF EACH MAP
NUMBER_OF_MAPS = len(MAPS)                                 #FINDING THE NUM-
    BER OF MAPS, TO EVALUATE THE PROBABILITY OF THE OCCURANCE OF A
    FLOOR TILE ON EACH BLOCK
START_I = 0                                                               #i-INDEX
    FROM WHICH MAP GENERATION, AFTER THE PLACEMENT OF THE FIRST
```

```
        TILE BEGINS
START_J = 0                                             #j-INDEX
        FROM WHICH MAP GENERATION, AFTER THE PLACEMENT OF THE FIRST
        TILE BEGINS
ROWS_LIMIT = 8
COLUMNS_LIMIT = 8
#######################################

ACCUMULATOR = []                                        #MATRIX TO
        STORE THE PROBABILITIES OF THE OCCURANCE OF A FLOOR TILE ON
        EACH BLOCK
NEW_MAP = []                                            #MATRIX TO
        STORE STATS OF THE GENERATED MAP

#INITIALIZING ACCUMULATOR AND NEW_MAP
print 'INITIALIZING NEW MAP______'

for i in range(0, ROWS_LIMIT):
        temp1 = []
        temp2 = []
        for j in range(0, COLUMNS_LIMIT):
                temp1.append(0.)
                temp2.append(999)
        ACCUMULATOR.append(temp1)
        NEW_MAP.append(temp2)

ACCUMULATOR = np.matrix(ACCUMULATOR)
NEW_MAP = np.matrix(NEW_MAP)

#######################################

print 'ANALYZING MAPS______'
for val in MAPS:

        current_map = MAPS_PATH+val                     #PATH TO CURRENT
        MAP
        CURRENT_MAP_MATRIX = []                         #TO STORE
        STATS OF THE CURRENT MAP: WHETHER A FLOOR TILE OCCURS ON A
        BLOCK, OR NOT

        #READING THE CURRENT MAP FILE
        with open(current_map,'rb') as csvfile:
                reader = csv.reader(csvfile,delimiter='	')
                temp = []
```

```
            for row in reader:
                for val in row:
                    temp.append(float(val))

                CURRENT_MAP_MATRIX.append(temp)
                temp = []

        CURRENT_MAP_MATRIX = np.matrix(CURRENT_MAP_MATRIX)
        CURRENT_MAP_MATRIX = CURRENT_MAP_MATRIX[1:-1,1:-1]

        #UPDATING THE ACCUMULATOR
        for i in range (0, ROWS_LIMIT):
            for j in range(0, COLUMNS_LIMIT):
                if (CURRENT_MAP_MATRIX[i, j] > 6):
                    ACCUMULATOR[i, j] += 1

ACCUMULATOR/=NUMBER_OF_MAPS                                      #FINDING  THE
        PROBABILITY OF A FLOOR TILE ON EACH BLOCK

#print(ACCUMULATOR)
print 'ACCUMULATED OBJECT DISTRIBUTION STATS______'

i = 0
j = 0
while not (0 in NEW_MAP):
        #print('Hi')
        NEW_MAP[i, j] = randomTileClass(ACCUMULATOR[i, j])

        if NEW_MAP[i, j] != 0:
            if (i == ROWS_LIMIT - 1) and (j == COLUMNS_LIMIT - 1):
                print('MAP FAILURE______')
                break

            elif (i < ROWS_LIMIT) and (j < COLUMNS_LIMIT):

                if i == ROWS_LIMIT:
                    i = 0
                    j += 1

                else:
                    i += 1
        else:
            START_I = i
            START_J = j
```

```
    #print (i, j)

#print(START_I, START_J)

for j in range(0, ROWS_LIMIT):
    for i in range(0, ROWS_LIMIT - 1):

        currentBlockValue = NEW_MAP[i, j]
        #print('LALALA')
        #print(currentBlockValue)

        leftNeighbour = (findLeftNeighbour(i,j))
        leftNeighbourValue = -1
        neighbourValues = []
        if leftNeighbour != -1:
            a,b = leftNeighbour
            leftNeighbourValue = NEW_MAP[a, b]

        while True:
            if (i < ROWS_LIMIT - 1):
                downNeighbourValue = randomTileClass(ACCUMULATOR[i, j])
            else:
                downNeighbourValue = -1


            if (j < COLUMNS_LIMIT - 1):
                rightNeighbourValue = randomTileClass(ACCUMULATOR[i, j])
            else:
                rightNeighbourValue = -1


            if currentBlockValue == 0:
                if (leftNeighbourValue == 0) or (downNeighbourValue == 0):
                    #print('REACHED')
                    break

            break

        if (rightNeighbourValue != -1):
            NEW_MAP[i, j+1] = rightNeighbourValue


        if (downNeighbourValue != -1):
            NEW_MAP[i+1, j] = downNeighbourValue
            """
            print('THIS')
            print(i, j)
            print(neighbourValues)
            """
#print(NEW_MAP)

#####################################

print 'SAVING MAP METADATA______'
np.savetxt("NEW_MAP.map", NEW_MAP, delimiter = '     ')


#print(MAP_OUT)


//main.cpp:
//C++ Header Files
#include <stdio.h>

//SDL Header Files
#include <SDL.h>

//Project Header Files
#include "globals.h"
#include "functions.h"

int main( int argc, char* args[] )
{
    if ( !InitLibraries() )
    {
        printf("\nFailed to Initialize!\n");
    }

    else
    {
        //The level tiles
        Tile* tileSet[ NUM_TILES ];

        //Load media
        if( !LoadMedia(tileSet) )
        {
            printf( "Failed to load media!\n" );
        }
```

```
else
{
    //Main loop flag
    bool quit = false;

    //DEBUG
    bool projectileAlive = false;

    //Event handler
    SDL_Event e;

    //Create Our Playable Character Objects
    Kid kid;
    Zombie zombie;
    LTimer deathTimer;

    //Animation Frame Data
    int kidFrameNum = 0;
    int zombieFrameNum = 0;
    int fireBallFrameNum = 0;


    //Player States
    playerstates kidState;
    playerstates zombieState;
    bool hit = false;

    //Fireball State
    projectilestates fireBallState;

    //colour modulation for highlighting collisions
    Uint8 kidR = 255;
    Uint8 kidG = 255;
    Uint8 kidB = 255;
    Uint8 zombieR = 255;
    Uint8 zombieG = 255;
    Uint8 zombieB = 255;

    //While application is running
    while( !quit )
    {
        //Handle events on queue
        while( SDL_PollEvent( &e ) != 0 )
        {
```

```
        if(e.type == SDL_QUIT)
        {
            quit = true;
        }

        //Handle Input For The Players
        kid.HandleEvent( e );
        zombie.HandleEvent( e );
    }

    //Get Current State of Players
    kidState = kid.GetState();
    zombieState = zombie.GetState();

    //Render characters
    kid.Move(tileSet, &kidR, &kidG, &kidB, kidState);
    zombie.Move(tileSet, &zombieR, &zombieG, &zombieB);

    if(projectileCreated)
    {
        //timer.start();
        fireBall = new FireBall(kid.GetXPos(),kid.GetYPos(),kid.GetAngle());
        projectileCreated = false;
        projectileAlive = true;
        //projectileAllowed = false;
    }

    //Update Player States
    if (PlayerCollision(&kid, &zombie))
    {
        kid.SetState(DEAD);
        kidHealth = 0;
        strcpy(kidHealthMessage, "Kid Health: ");
        snprintf(kidHealthStatus,5,"%d",kidHealth);
        strcat(kidHealthMessage,kidHealthStatus);
    }

    if(projectileAlive)
    {
        fireBall->Move();
        if(!FireBallCollision(&kid, &zombie, tileSet))
        {
            projectileAlive = true;
        }
    else
    {
        if(ProjectileImpact(&zombie))
        {
            zombieHealth -= 1000;
            strcpy(zombieHealthMessage, "Zombie Health: ");
            snprintf(zombieHealthStatus,5,"%d",zombieHealth);
            strcat(zombieHealthMessage,zombieHealthStatus);
            deathTimer.start();
            hit = true;

            if(zombieHealth == 0)
            {
               zombie.SetState(DEAD);
            }
        }

        projectileAlive = false;
        delete fireBall;
    }
}

if (deathTimer.getTicks() >= 500)
{
    hit = false;
}

if(hit)
{
    zombieG = 32;
}

else
{
    zombieG = 255;
}


//Clear screen
SDL_SetRenderDrawColor( windowRenderer, 0x00, 0x00, 0x00, 0x00 );
SDL_RenderClear( windowRenderer );

//Render level
```

```
for(int i=0; i<NUM_TILES; i++)
{
   tileSet[i]->Render();
}

kid.Animate(kidFrameNum, kidR, kidG, kidB, kidState);
zombie.Animate(zombieFrameNum, zombieState, zombieR, zombieG, zombieB);

if(projectileAlive)
{
   if(!FireBallCollision(&kid, &zombie, tileSet))
   {
       fireBall -> Animate(fireBallFrameNum, fireBallState);
   }

   else
   {
      fireBall -> Animate(fireBallFrameNum, fireBallState);
   }
}

//Go to next frame
kidFrameNum++;
zombieFrameNum++;
fireBallFrameNum++;

//Cycle animations
if( kidFrameNum/4 >= KID_MOVE_ANIMATION_FRAMES )
{
   kidFrameNum = 0;
}

if( zombieFrameNum/4 >= ZOMBIE_MOVE_ANIMATION_FRAMES )
{
   zombieFrameNum = 0;
}

if( fireBallFrameNum/24 >= FIREBALL_TRAVEL_ANIMATION_FRAMES )
{
   fireBallFrameNum = 0;
}

zombieHealthBar.LoadFromRenderedText(zombieHealthMessage, textColor);
zombieHealthBar.Render(100,100);
```

```
                kidHealthBar.LoadFromRenderedText(kidHealthMessage, textColor);
                kidHealthBar.Render(300,500);

                printf("\n%d",projectileCreated);


                //Update screen
                SDL_RenderPresent( windowRenderer );
            }
        }
        //Free resources and Close SDL
        Close( tileSet );
    }
    return 0;


//  functions.cpp
#include "functions.h"
#include "globals.h"
#include <SDL_image.h>
#include <fstream>

bool InitLibraries ( void )
{
    //Initialization flag
    bool success = true;

    //Initialize SDL
    if( SDL_Init( SDL_INIT_VIDEO ) < 0 )
    {
        printf( "SDL could not initialize! SDL Error: %s\n", SDL_GetError() );
        success = false;
    }
    else
    {
        //Set texture filtering to linear
        if( !SDL_SetHint( SDL_HINT_RENDER_SCALE_QUALITY, "1" ) )
        {
            printf( "Warning: Linear texture filtering not enabled!" );
        }

        //Create window
        mainWindow = SDL_CreateWindow( "SDL Tutorial", SDL_WINDOWPOS_UNDE-
FINED, SDL_WINDOWPOS_UNDEFINED, SCREEN_WIDTH, SCREEN_HEIGHT,
SDL_WINDOW_SHOWN );
        if( mainWindow == NULL )
        {
            printf( "Window could not be created! SDL Error: %s\n", SDL_GetError() );
            success = false;
        }
        else
        {
            //Create vsynced renderer for window
            windowRenderer = SDL_CreateRenderer( mainWindow, -1,
SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC );
            if( windowRenderer == NULL )
            {
                printf( "Renderer could not be created! SDL Error: %s\n", SDL_GetError() );
                success = false;
            }
            else
            {
                //Initialize renderer color
                SDL_SetRenderDrawColor( windowRenderer, 0xFF, 0xFF, 0xFF, 0xFF );

                //Initialize PNG loading
                int imgFlags = IMG_INIT_PNG;
                if( !( IMG_Init( imgFlags ) & imgFlags ) )
                {
                    printf( "SDL_image could not initialize! SDL_image Error: %s\n",
IMG_GetError() );
                    success = false;
                }
            }
        }
        //Initialize SDL_ttf
        if( TTF_Init() == -1 )
        {
            printf( "SDL_ttf could not initialize! SDL_ttf Error: %s\n", TTF_GetError() );
            success = false;
        }
    }

    return success;
}

bool LoadMedia( Tile* tiles[])
```

```
{
   //Loading success flag
   bool success = true;

   //Load kid texture
   if( !kidSpriteSheet.LoadFromFile( "Assets/Kid/Kid.png" ))
   {
      printf( "Failed to load kid texture!\n" );
      success = false;
   }

   else
   {
      //Set sprite clips
      kidMove[ 0 ].x =  64;
      kidMove[ 0 ].y =   0;
      kidMove[ 0 ].w =  64;
      kidMove[ 0 ].h =  64;

      kidMove[ 1 ].x = 128;
      kidMove[ 1 ].y =   0;
      kidMove[ 1 ].w =  64;
      kidMove[ 1 ].h =  64;

      kidMove[ 2 ].x = 192;
      kidMove[ 2 ].y =   0;
      kidMove[ 2 ].w =  64;
      kidMove[ 2 ].h =  64;

      kidMove[ 3 ].x = 256;
      kidMove[ 3 ].y =   0;
      kidMove[ 3 ].w =  64;
      kidMove[ 3 ].h =  64;

      kidDeath[ 0 ].x =   0;
      kidDeath[ 0 ].y =   0;
      kidDeath[ 0 ].w =  64;
      kidDeath[ 0 ].h =  64;
   }

   if( !fireBallSpriteSheet.LoadFromFile("Assets/FireBall/Fireball.png"))
   {
      printf( "Failed to load FireBall textures!\n" );
      success = false;
```

```
}

else
{
    //Set sprite clips
    fireBallTravel[ 0 ].x =   0;
    fireBallTravel[ 0 ].y =   0;
    fireBallTravel[ 0 ].w =  64;
    fireBallTravel[ 0 ].h =  64;

    fireBallTravel[ 1 ].x =  64;
    fireBallTravel[ 1 ].y =   0;
    fireBallTravel[ 1 ].w =  64;
    fireBallTravel[ 1 ].h =  64;

    fireBallTravel[ 2 ].x = 128;
    fireBallTravel[ 2 ].y =   0;
    fireBallTravel[ 2 ].w =  64;
    fireBallTravel[ 2 ].h =  64;

    fireBallTravel[ 3 ].x = 192;
    fireBallTravel[ 3 ].y =   0;
    fireBallTravel[ 3 ].w =  64;
    fireBallTravel[ 3 ].h =  64;

    fireBallTravel[ 4 ].x = 256;
    fireBallTravel[ 4 ].y =   0;
    fireBallTravel[ 4 ].w =  64;
    fireBallTravel[ 4 ].h =  64;

    fireBallTravel[ 5 ].x = 320;
    fireBallTravel[ 5 ].y =   0;
    fireBallTravel[ 5 ].w =  64;
    fireBallTravel[ 5 ].h =  64;

    fireBallImpact[ 0 ].x = 384;
    fireBallImpact[ 0 ].y =   0;
    fireBallImpact[ 0 ].w =  64;
    fireBallImpact[ 0 ].h =  64;

    fireBallImpact[ 1 ].x = 448;
    fireBallImpact[ 1 ].y =   0;
    fireBallImpact[ 1 ].w =  64;
    fireBallImpact[ 1 ].h =  64;
```

```
        fireBallImpact[ 2 ].x = 512;
        fireBallImpact[ 2 ].y =   0;
        fireBallImpact[ 2 ].w =  64;
        fireBallImpact[ 2 ].h =  64;

        fireBallImpact[ 3 ].x = 576;
        fireBallImpact[ 3 ].y =   0;
        fireBallImpact[ 3 ].w =  64;
        fireBallImpact[ 3 ].h =  64;

        fireBallImpact[ 4 ].x = 640;
        fireBallImpact[ 4 ].y =   0;
        fireBallImpact[ 4 ].w =  64;
        fireBallImpact[ 4 ].h =  64;

        fireBallImpact[ 5 ].x = 704;
        fireBallImpact[ 5 ].y =   0;
        fireBallImpact[ 5 ].w =  64;
        fireBallImpact[ 5 ].h =  64;

        fireBallImpact[ 6 ].x = 768;
        fireBallImpact[ 6 ].y =   0;
        fireBallImpact[ 6 ].w =  64;
        fireBallImpact[ 6 ].h =  64;

        fireBallImpact[ 7 ].x = 832;
        fireBallImpact[ 7 ].y =   0;
        fireBallImpact[ 7 ].w =  64;
        fireBallImpact[ 7 ].h =  64;

        fireBallImpact[ 8 ].x = 896;
        fireBallImpact[ 8 ].y =   0;
        fireBallImpact[ 8 ].w =  64;
        fireBallImpact[ 8 ].h =  64;
    }

    //Load kid texture
    if( !tilesSpriteSheet.LoadFromFile( "Assets/Tiles/Tiles.png" ))
    {
        printf( "Failed to load floor textures!\n" );
        success = false;
    }
```

```
if (!SetTiles(tiles))
{
    printf ("\nFailed To Load Tile Set!");
    success = false;
}

else
{
    //Set sprite clips
    tilesRects[ FLOOR_1 ].x =   0;
    tilesRects[ FLOOR_1 ].y =   0;
    tilesRects[ FLOOR_1 ].w =  64;
    tilesRects[ FLOOR_1 ].h =  64;

    tilesRects[ FLOOR_2 ].x =  64;
    tilesRects[ FLOOR_2 ].y =   0;
    tilesRects[ FLOOR_2 ].w =  64;
    tilesRects[ FLOOR_2 ].h =  64;

    tilesRects[ FLOOR_3 ].x = 128;
    tilesRects[ FLOOR_3 ].y =   0;
    tilesRects[ FLOOR_3 ].w =  64;
    tilesRects[ FLOOR_3 ].h =  64;

    tilesRects[ FLOOR_4 ].x = 192;
    tilesRects[ FLOOR_4 ].y =   0;
    tilesRects[ FLOOR_4 ].w =  64;
    tilesRects[ FLOOR_4 ].h =  64;

    tilesRects[ FLOOR_5 ].x = 256;
    tilesRects[ FLOOR_5 ].y =   0;
    tilesRects[ FLOOR_5 ].w =  64;
    tilesRects[ FLOOR_5 ].h =  64;

    tilesRects[ FLOOR_6 ].x = 320;
    tilesRects[ FLOOR_6 ].y =   0;
    tilesRects[ FLOOR_6 ].w =  64;
    tilesRects[ FLOOR_6 ].h =  64;

    tilesRects[ FLOOR_7 ].x = 384;
    tilesRects[ FLOOR_7 ].y =   0;
    tilesRects[ FLOOR_7 ].w =  64;
    tilesRects[ FLOOR_7 ].h =  64;
```

```
tilesRects[ FLOOR_8 ].x = 448;
tilesRects[ FLOOR_8 ].y =   0;
tilesRects[ FLOOR_8 ].w =  64;
tilesRects[ FLOOR_8 ].h =  64;

tilesRects[ FLOOR_9 ].x = 512;
tilesRects[ FLOOR_9 ].y =   0;
tilesRects[ FLOOR_9 ].w =  64;
tilesRects[ FLOOR_9 ].h =  64;

tilesRects[ LAVA_MIDDLE ].x = 576;
tilesRects[ LAVA_MIDDLE ].y =   0;
tilesRects[ LAVA_MIDDLE ].w =  64;
tilesRects[ LAVA_MIDDLE ].h =  64;

tilesRects[ LAVA_EAST ].x = 640;
tilesRects[ LAVA_EAST ].y =   0;
tilesRects[ LAVA_EAST ].w =  64;
tilesRects[ LAVA_EAST ].h =  64;

tilesRects[ LAVA_NORTH ].x = 704;
tilesRects[ LAVA_NORTH ].y =   0;
tilesRects[ LAVA_NORTH ].w =  64;
tilesRects[ LAVA_NORTH ].h =  64;

tilesRects[ LAVA_NORTH_EAST ].x = 768;
tilesRects[ LAVA_NORTH_EAST ].y =   0;
tilesRects[ LAVA_NORTH_EAST ].w =  64;
tilesRects[ LAVA_NORTH_EAST ].h =  64;

tilesRects[ LAVA_NORTH_WEST ].x = 832;
tilesRects[ LAVA_NORTH_WEST ].y =   0;
tilesRects[ LAVA_NORTH_WEST ].w =  64;
tilesRects[ LAVA_NORTH_WEST ].h =  64;

tilesRects[ LAVA_SOUTH ].x = 896;
tilesRects[ LAVA_SOUTH ].y =   0;
tilesRects[ LAVA_SOUTH ].w =  64;
tilesRects[ LAVA_SOUTH ].h =  64;

tilesRects[ LAVA_SOUTH_EAST ].x = 960;
tilesRects[ LAVA_SOUTH_EAST ].y =   0;
tilesRects[ LAVA_SOUTH_EAST ].w =  64;
tilesRects[ LAVA_SOUTH_EAST ].h =  64;
```

```
    tilesRects[ LAVA_SOUTH_WEST ].x = 1024;
    tilesRects[ LAVA_SOUTH_WEST ].y =   0;
    tilesRects[ LAVA_SOUTH_WEST ].w =  64;
    tilesRects[ LAVA_SOUTH_WEST ].h =  64;

    tilesRects[ LAVA_WEST ].x = 1088;
    tilesRects[ LAVA_WEST ].y =   0;
    tilesRects[ LAVA_WEST ].w =  64;
    tilesRects[ LAVA_WEST ].h =  64;
}

//Load zombie texture
if( !zombieSpriteSheet.LoadFromFile( "Assets/Zombie/Zombie.png" ) )
{
    printf( "Failed to load zombie texture!\n" );
    success = false;
}

else
{
    zombieMove[ 0 ].x =   0;
    zombieMove[ 0 ].y =   0;
    zombieMove[ 0 ].w =  64;
    zombieMove[ 0 ].h =  64;

    zombieMove[ 1 ].x =  64;
    zombieMove[ 1 ].y =   0;
    zombieMove[ 1 ].w =  64;
    zombieMove[ 1 ].h =  64;

    zombieMove[ 2 ].x = 128;
    zombieMove[ 2 ].y =   0;
    zombieMove[ 2 ].w =  64;
    zombieMove[ 2 ].h =  64;

    zombieMove[ 3 ].x = 192;
    zombieMove[ 3 ].y =   0;
    zombieMove[ 3 ].w =  64;
    zombieMove[ 3 ].h =  64;

    zombieDeath[ 0 ].x = 320;
    zombieDeath[ 0 ].y =   0;
    zombieDeath[ 0 ].w =  64;
```

```
        zombieDeath[ 0 ].h =  64;
    }

    //Open the font
    gFont = TTF_OpenFont( "Assets/INVASION2000.ttf", 28 );
    if( gFont == NULL )
    {
        printf( "Failed to load lazy font! SDL_ttf Error: %s\n", TTF_GetError() );
        success = false;
    }

    return success;
}

void Close( Tile* tiles[])
{
    //Deallocate tiles
    for( int i = 0; i < NUM_TILES; i++ )
    {
        if( tiles[ i ] == NULL )
        {
            delete tiles[ i ];
            tiles[ i ] = NULL;
        }
    }

    //Free loaded images
    kidSpriteSheet.Free();
    zombieSpriteSheet.Free();
    tilesSpriteSheet.Free();
    fireBallSpriteSheet.Free();

    //Destroy window
    SDL_DestroyRenderer( windowRenderer );
    SDL_DestroyWindow( mainWindow );
    mainWindow = NULL;
    windowRenderer = NULL;

    //Quit SDL subsystems
    IMG_Quit();
    SDL_Quit();
    TTF_Quit();
}
```

```
bool CheckCollision( SDL_Rect a, SDL_Rect b )
{
    //The sides of the rectangles
    int leftA, leftB;
    int rightA, rightB;
    int topA, topB;
    int bottomA, bottomB;

    //Calculate the sides of rect A
    leftA = a.x;
    rightA = a.x + a.w;
    topA = a.y;
    bottomA = a.y + a.h;

    //Calculate the sides of rect B
    leftB = b.x;
    rightB = b.x + b.w;
    topB = b.y;
    bottomB = b.y + b.h;

    //If any of the sides from A are outside of B
    if( bottomA <= topB )
    {
        return false;
    }

    if( topA >= bottomB )
    {
        return false;
    }

    if( rightA <= leftB )
    {
        return false;
    }

    if( leftA >= rightB )
    {
        return false;
    }

    //If none of the sides from A are outside B
    return true;
}
```

```
bool SetTiles (Tile* tiles[])
{
    bool tilesLoaded = true;
    int x=0, y=0;

    //We'll read from a file called level1.map
    std::ifstream map;
    map.open("Assets/Levels/level1.map");

    // If we couldn't open the output file stream for reading
    if (!map)
    {
        // Print an error and exit
        printf("\nThe Map File Could Not Be Opened!\n\n");
        tilesLoaded = false;
    }

    else
    {
        // While there's still stuff left to read
        for(int i=0; i<NUM_TILES; i++)
        {
            //Determines what kind of tile will be made
            int tileType = -1;

            //Read tile from map file
            map >> tileType;

            //If the was a problem in reading the map
            if( map.fail() )
            {
                //Stop loading map
                printf( "Error loading map: Unexpected end of file!\n" );
                tilesLoaded = false;
                break;
            }

            //If the number is a valid tile number
            if( ( tileType >= 0 ) && ( tileType < NUM_TILES ) )
            {
                tiles[ i ] = new Tile( x, y, tileType );

                switch(tileType)
```

```
{
    case FLOOR_1:   tiles[ i ]->texture = 0;
                    break;

    case FLOOR_2:   tiles[ i ]->texture = 1;
                    break;

    case FLOOR_3:   tiles[ i ]->texture = 2;
                    break;

    case FLOOR_4:   tiles[ i ]->texture = 3;
                    break;

    case FLOOR_5:   tiles[ i ]->texture = 4;
                    break;

    case FLOOR_6:   tiles[ i ]->texture = 5;
                    break;

    case FLOOR_7:   tiles[ i ]->texture = 6;
                    break;

    case FLOOR_8:   tiles[ i ]->texture = 7;
                    break;

    case FLOOR_9:   tiles[ i ]->texture = 8;
                    break;

    case LAVA_MIDDLE:   tiles[ i ]->texture = 9;
                        break;

    case LAVA_EAST:     tiles[ i ]->texture = 10;
                        break;

    case LAVA_NORTH:    tiles[ i ]->texture = 11;
                        break;

    case LAVA_NORTH_EAST:tiles[ i ]->texture = 12;
                        break;

    case LAVA_NORTH_WEST:tiles[ i ]->texture = 13;
                        break;

    case LAVA_SOUTH:    tiles[ i ]->texture = 14;
```

```
                        break;

                case LAVA_SOUTH_EAST:tiles[ i ]->texture = 15;
                        break;

                case LAVA_SOUTH_WEST:tiles[ i ]->texture = 16;
                        break;

                case LAVA_WEST:     tiles[ i ]->texture = 17;
                        break;
              }
            }
            //If we don't recognize the tile type
            else
            {
              //Stop loading map
              printf( "Error loading map: Invalid tile type at %d!\n", i );
              tilesLoaded = false;
              break;
            }

            //Move to next tile spot
            x += TILE_WIDTH;

            //If we've gone too far
            if( x >= SCREEN_WIDTH )
            {
              //Move back
              x = 0;

              //Move to the next row
              y += TILE_HEIGHT;
            }
          }
        }
        //Close the file
        map.close();

        //If the map was loaded fine
        return tilesLoaded;

}

bool TouchesWall( SDL_Rect box, Tile* tiles[] )
```

```
{
   //Go through the tiles
   for( int i = 0; i < NUM_TILES; ++i )
   {
      //If the tile is a wall type tile
      if( ( tiles[ i ]->GetType() >= LAVA_MIDDLE ) && ( tiles[ i ]->GetType() <=
LAVA_WEST ) )
      {
         //If the collision box touches the wall tile
         if( CheckCollision( box, tiles[ i ]->GetRect() ) )
         {
            return true;
         }
      }
   }

   //If no wall tiles were touched
   return false;
}

double FindAngle (int x1, int x2, int y1, int y2)
{
   double deltaX, deltaY;
   double angle;

   deltaX = double(x2 - x1);
   deltaY = double(y2 - y1);

   angle = atan2(deltaY, deltaX) * (180/PI);

   return angle;
}


bool PlayerCollision( Kid* kid, Zombie* zombie )
{
   SDL_Rect kidRect, zombieRect;

   kidRect = kid -> GetRect();
   zombieRect = zombie -> GetRect();

   return CheckCollision(kidRect, zombieRect);
}
```

```
bool FireBallCollision( Kid* kid, Zombie* zombie, Tile* tiles[] )
{
   SDL_Rect kidRect, zombieRect, fireBallRect;

   kidRect = kid -> GetRect();
   zombieRect = zombie -> GetRect();
   fireBallRect = fireBall -> GetRect();

   if ( CheckCollision(fireBallRect, zombieRect) || TouchesWall(fireBallRect, tiles))
   {
      return true;
   }

   else
   {
      return false;
   }
}

bool ProjectileImpact(Zombie* zombie)
{
   SDL_Rect zombieRect, fireBallRect;

   zombieRect = zombie -> GetRect();
   fireBallRect = fireBall -> GetRect();

   if ( CheckCollision(fireBallRect, zombieRect))
   {
      return true;
   }

   else
   {
      return false;
   }
}

//fireball.cpp
#include "fireball.h"
#include "globals.h"

FireBall::FireBall(int x, int y, double angle)
{
   posX = x;
```

```
    posY = y;
    collider.x = x+20;
    collider.y = y+20;
    collider.w = FIREBALL_WIDTH-40;
    collider.h = FIREBALL_HEIGHT-40;

    this -> angle = angle;
}

void FireBall::Animate(int frameNum, projectilestates state)
{
    //Animation Data
    SDL_Rect* currentFrame;

    switch(state)
    {
        case TRAVEL: currentFrame = &fireBallTravel[frameNum/24]; break;
        case IMPACT: currentFrame = &fireBallImpact[frameNum]; break;
    }

    fireBallSpriteSheet.Render(posX,posY,currentFrame,angle,NULL,SDL_FLIP_NONE);
}

SDL_Rect FireBall::GetRect()
{
    return collider;
}

void FireBall::Move()
{
    velX = FIREBALL_VEL * sin(angle*(PI/180));
    velY = FIREBALL_VEL * -cos(angle*(PI/180));

    posX += velX;
    collider.x += velX;

    posY += velY;
    collider.y += velY;
}



//constant.h
#ifndef CONSTANTS_H
```

```
#define CONSTANTS_H

#define PI 3.1415

//Screen Window Resolution
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 640;

//Tile Constants
const int NUM_TILES = 100;
const int TILE_WIDTH = 64;
const int TILE_HEIGHT = 64;
enum tiletype
{
   FLOOR_1,
   FLOOR_2,
   FLOOR_3,
   FLOOR_4,
   FLOOR_5,
   FLOOR_6,
   FLOOR_7,
   FLOOR_8,
   FLOOR_9,
   LAVA_MIDDLE,
   LAVA_EAST,
   LAVA_NORTH,
   LAVA_NORTH_EAST,
   LAVA_NORTH_WEST,
   LAVA_SOUTH,
   LAVA_SOUTH_EAST,
   LAVA_SOUTH_WEST,
   LAVA_WEST
};

//Player States
enum playerstates
{
   DEAD,
   IN_PLAY,
   SHOOTING,
   HURT
};

//Game States
```

```
enum gamestates
{
   START_SCREEN,
   POSITION_SCREEN,
   IN_GAME,
   GAME_OVER
};

//Projectile States
enum projectilestates
{
   TRAVEL,
   IMPACT
};

//Zombie Properties
const int ZOMBIE_WIDTH = 64;
const int ZOMBIE_HEIGHT = 64;
const int ZOMBIE_VEL = 5;
const int ZOMBIE_DEATH_ANIMATION_FRAMES = 1;
const int ZOMBIE_MOVE_ANIMATION_FRAMES = 4;

//Kid Properties
const int KID_WIDTH = 64;
const int KID_HEIGHT = 64;
const int KID_VEL = 5;
const int KID_DEATH_ANIMATION_FRAMES = 1;
const int KID_MOVE_ANIMATION_FRAMES = 4;
const int KID_SHOOT_ANIMATION_FRAMES = 2;

//Fireball Properties
const int FIREBALL_WIDTH = 64;
const int FIREBALL_HEIGHT = 64;
const int FIREBALL_VEL = 9;
const int FIREBALL_TRAVEL_ANIMATION_FRAMES = 6;
const int FIREBALL_IMPACT_ANIMATION_FRAMES = 9;

#endif /* constants_h */


//globals.cpp
#include <SDL.h>
#include "texture.h"
#include "constants.h"
```

```
#include <SDL_ttf.h>
#include "fireball.h"
#include "timer.h"
#include "globals.h"

//Window That is Being Rendered To
SDL_Window* mainWindow = NULL;

//The window renderer
SDL_Renderer* windowRenderer = NULL;

//Animation Data For Kid
Texture kidSpriteSheet;
SDL_Rect kidShoot[KID_SHOOT_ANIMATION_FRAMES];
SDL_Rect kidMove[KID_MOVE_ANIMATION_FRAMES];
SDL_Rect kidDeath[KID_DEATH_ANIMATION_FRAMES];

//Animation Data For Zombie
Texture zombieSpriteSheet;
SDL_Rect zombieMove[ZOMBIE_MOVE_ANIMATION_FRAMES];
SDL_Rect zombieDeath[ZOMBIE_DEATH_ANIMATION_FRAMES];

//Spritesheets and Data For Tiles
Texture tilesSpriteSheet;
SDL_Rect tilesRects[NUM_TILES];

//Spritesheets and Data for Fireball
Texture fireBallSpriteSheet;
SDL_Rect fireBallTravel[FIREBALL_TRAVEL_ANIMATION_FRAMES];
SDL_Rect fireBallImpact[FIREBALL_IMPACT_ANIMATION_FRAMES];

bool clickEnabled = true;
bool projectileCreated = false;

FireBall* fireBall = NULL;
int kidHealth = 4000;
int zombieHealth = 4000;

LTimer projectileTimer;

//Globally used font
TTF_Font *gFont = NULL;

//Rendered texture
```

```
Texture zombieHealthBar;
Texture kidHealthBar;
SDL_Color textColor = {255, 255, 255};

char kidHealthMessage[20] = "Kid Health: ";
char zombieHealthMessage[20] = "Zombie Health: ";

char kidHealthStatus[5] = "4000";
char zombieHealthStatus[5] = "4000";



//texture.cpp
#include "texture.h"
#include <SDL_image.h>
#include <SDL_ttf.h>
#include "globals.h"

Texture::Texture()
{
   //Initialize
   mTexture = NULL;
   mWidth = 0;
   mHeight = 0;
}

Texture::~Texture()
{
   //Deallocate
   Free();
}

bool Texture::LoadFromFile( std::string path )
{
   //Get rid of preexisting texture
   Free();

   //The final texture
   SDL_Texture* newTexture = NULL;

   //Load image at specified path
   SDL_Surface* loadedSurface = IMG_Load( path.c_str() );
   if( loadedSurface == NULL )
   {
        printf( "Unable to load image %s! SDL_image Error: %s\n", path.c_str(), IMG_GetEr-
ror() );
    }
    else
    {
        //Color key image
        SDL_SetColorKey( loadedSurface, SDL_TRUE, SDL_MapRGB( loadedSurface->for-
mat, 0, 0xFF, 0xFF ) );

        //Create texture from surface pixels
        newTexture = SDL_CreateTextureFromSurface( windowRenderer, loadedSurface );
        if( newTexture == NULL )
        {
            printf( "Unable to create texture from %s! SDL Error: %s\n", path.c_str(),
SDL_GetError() );
        }
        else
        {
            //Get image dimensions
            mWidth = loadedSurface->w;
            mHeight = loadedSurface->h;
        }

        //Get rid of old loaded surface
        SDL_FreeSurface( loadedSurface );
    }

    //Return success
    mTexture = newTexture;
    return mTexture != NULL;
}

//#ifdef _SDL_TTF_H
bool Texture::LoadFromRenderedText( std::string textureText, SDL_Color textColor )
{
    //Get rid of preexisting texture
    Free();

    //Render text surface
    SDL_Surface* textSurface = TTF_RenderText_Solid( gFont, textureText.c_str(), text-
Color );
    if( textSurface != NULL )
    {
        //Create texture from surface pixels
        mTexture = SDL_CreateTextureFromSurface( windowRenderer, textSurface );
        if( mTexture == NULL )
        {
            printf( "Unable to create texture from rendered text! SDL Error: %s\n", SDL_GetEr-
ror() );
        }
        else
        {
            //Get image dimensions
            mWidth = textSurface->w;
            mHeight = textSurface->h;
        }

        //Get rid of old surface
        SDL_FreeSurface( textSurface );
    }
    else
    {
        printf( "Unable to render text surface! SDL_ttf Error: %s\n", TTF_GetError() );
    }


    //Return success
    return mTexture != NULL;
}
//#endif

void Texture::Free()
{
    //Free texture if it exists
    if( mTexture != NULL )
    {
        SDL_DestroyTexture( mTexture );
        mTexture = NULL;
        mWidth = 0;
        mHeight = 0;
    }
}

void Texture::SetColor( Uint8 red, Uint8 green, Uint8 blue )
{
    //Modulate texture rgb
    SDL_SetTextureColorMod( mTexture, red, green, blue );
}
```

```
void Texture::SetBlendMode( SDL_BlendMode blending )
{
   //Set blending function
   SDL_SetTextureBlendMode( mTexture, blending );
}

void Texture::SetAlpha( Uint8 alpha )
{
   //Modulate texture alpha
   SDL_SetTextureAlphaMod( mTexture, alpha );
}

void Texture::Render( int x, int y, SDL_Rect* clip, double angle, SDL_Point* center,
SDL_RendererFlip flip )
{
   //Set rendering space and render to screen
   SDL_Rect renderQuad = { x, y, mWidth, mHeight };

   //Set clip rendering dimensions
   if( clip != NULL )
   {
      renderQuad.w = clip->w;
      renderQuad.h = clip->h;
   }

   //Render to screen
   SDL_RenderCopyEx( windowRenderer, mTexture, clip, &renderQuad, angle, center,
flip );
}

int Texture::GetWidth()
{
   return mWidth;
}

int Texture::GetHeight()
{
   return mHeight;
}


//timer.cpp
#include "timer.h"
```

```
LTimer::LTimer()
{
   //Initialize the variables
   mStartTicks = 0;
   mPausedTicks = 0;

   mPaused = false;
   mStarted = false;
}

void LTimer::start()
{
   //Start the timer
   mStarted = true;

   //Unpause the timer
   mPaused = false;

   //Get the current clock time
   mStartTicks = SDL_GetTicks();
   mPausedTicks = 0;
}

void LTimer::stop()
{
   //Stop the timer
   mStarted = false;

   //Unpause the timer
   mPaused = false;

   //Clear tick variables
   mStartTicks = 0;
   mPausedTicks = 0;
}

void LTimer::pause()
{
   //If the timer is running and isn't already paused
   if( mStarted && !mPaused )
   {
      //Pause the timer
      mPaused = true;
```

```
        //Calculate the paused ticks
        mPausedTicks = SDL_GetTicks() - mStartTicks;
        mStartTicks = 0;
    }
}

void LTimer::unpause()
{
    //If the timer is running and paused
    if( mStarted && mPaused )
    {
        //Unpause the timer
        mPaused = false;

        //Reset the starting ticks
        mStartTicks = SDL_GetTicks() - mPausedTicks;

        //Reset the paused ticks
        mPausedTicks = 0;
    }
}

Uint32 LTimer::getTicks()
{
    //The actual timer time
    Uint32 time = 0;

    //If the timer is running
    if( mStarted )
    {
        //If the timer is paused
        if( mPaused )
        {
            //Return the number of ticks when the timer was paused
            time = mPausedTicks;
        }
        else
        {
            //Return the current time minus the start time
            time = SDL_GetTicks() - mStartTicks;
        }
    }
    return time;
}

bool LTimer::isStarted()
{
    //Timer is running and paused or unpaused
    return mStarted;
}

bool LTimer::isPaused()
{
    //Timer is running and paused
    return mPaused && mStarted;
}



//tile.h
#ifndef tile_h
#define tile_h

#include <SDL.h>

class Tile
{
public:
    //Constructor
    Tile(int x, int y, int tileType);

    //Draws The Tile
    void Render();

    //Returns The Tile Collider, for Collision Detection
    SDL_Rect GetRect();

    //Returns the Tile Type
    int GetType();

    //A Texture Index Associated With The Tile
    int texture;

private:
    //Tile Collision Box
    SDL_Rect collider;
    //X and Y Position of the Tile
    int posX, posY;

    //The Type of Tile
    int tileType;

    //RGB Color Values, for Color Modulation
    Uint8 r,g,b;
};

#endif /* tile_hpp */


//players.cpp
#include "players.h"
#include "functions.h"
#include "globals.h"
#include "constants.h"

int Player::GetXPos()
{
    return posX;
}

int Player::GetYPos()
{
    return posY;
}

Kid::Kid()
{
    //Initialize the offsets
    posX = 100;
    posY = 100;

    //Set collision box dimension
    collider.w = KID_WIDTH-40;
    collider.h = KID_HEIGHT-40;
    collider.x = posX+20;
    collider.y = posY+20;

    //Initialize the velocity
    velX = 0;
    velY = 0;

    //Initialize the angle
    angle = 0;

    //Initialize Default State
    state = IN_PLAY;
}

void Kid::HandleEvent( SDL_Event& e)
{
    int x,y;

    SDL_GetMouseState( &x, &y );

    angle = FindAngle(y, posY, posX, x);

    //If a key was pressed
    if( e.type == SDL_KEYDOWN && e.key.repeat == 0 )
    {
        //Adjust the velocity
        switch( e.key.keysym.sym )
        {
            case SDLK_w: velY -= KID_VEL; break;
            case SDLK_s: velY += KID_VEL; break;
            case SDLK_a: velX -= KID_VEL; break;
            case SDLK_d: velX += KID_VEL; break;
        }
    }
    //If a key was released
    else if( e.type == SDL_KEYUP && e.key.repeat == 0 )
    {
        //Adjust the velocity
        switch( e.key.keysym.sym )
        {
            case SDLK_w: velY += KID_VEL; break;
            case SDLK_s: velY -= KID_VEL; break;
            case SDLK_a: velX += KID_VEL; break;
            case SDLK_d: velX -= KID_VEL; break;
        }
    }


    else if(e.type == SDL_MOUSEBUTTONDOWN && clickEnabled)
    {
        projectileCreated = true;
        clickEnabled = false;
        projectileTimer.start();

    }

    if(!clickEnabled && projectileTimer.getTicks()<=1000)
    {
        SDL_EventState(SDL_MOUSEBUTTONDOWN, SDL_DISABLE);
    }

    else
    {
        projectileTimer.stop();
        clickEnabled = true;
        SDL_EventState(SDL_MOUSEBUTTONDOWN, SDL_ENABLE);
    }
}

void Kid::Move( Tile* tiles[], Uint8* r, Uint8* g, Uint8* b, playerstates state )
{
    //Move the Kid left or right
    posX += velX;
    collider.x += velX;


    //If the Kid collided or went too far to the left or right
    if( ( posX < 0 ) || ( posX + KID_WIDTH > SCREEN_WIDTH ))
    {
        //Move back
        posX -= velX;
        collider.x -= velX;
    }

    if(TouchesWall(collider, tiles) )
    {
        //Move back
        posX -= velX;
        collider.x -= velX;
        *g = 32;
        kidHealth -= 25;
        strcpy(kidHealthMessage, "Kid Health: ");
        snprintf(kidHealthStatus,5,"%d",kidHealth);
        strcat(kidHealthMessage,kidHealthStatus);
        if(kidHealth == 0)
        {
            this->SetState(DEAD);
        }
    }

    else
    {
        *g = 255;
    }

    //Move the Kid up or down
    posY += velY;
    collider.y += velY;;

    //If the Kid collided or went too far up or down
    if( ( posY < 0 ) || ( posY + KID_HEIGHT > SCREEN_HEIGHT ) )
    {
        //Move back
        posY -= velY;
        collider.y -= velY;
    }

    if(TouchesWall(collider, tiles) )
    {
        //Move back
        posY -= velY;
        collider.y -= velY;
        *g = 32;
        kidHealth -= 25;
        strcpy(kidHealthMessage, "Kid Health: ");
        snprintf(kidHealthStatus,5,"%d",kidHealth);
        strcat(kidHealthMessage,kidHealthStatus);
        if(kidHealth == 0)
        {
            this->SetState(DEAD);
        }
    }
}

void Kid::Animate(int frameNum, Uint8 r, Uint8 g, Uint8 b, playerstates state)
{
    //Animation Data
    SDL_Rect* currentFrame;

    switch (state)
    {
        case IN_PLAY:
            if(velX !=0 || velY !=0)
                currentFrame = &kidMove[frameNum/4];

            else
                currentFrame = &kidMove[0];
            break;
        case DEAD:
            currentFrame = &kidDeath[0];
            break;
        default:
            if(velX !=0 || velY !=0)
                currentFrame = &kidMove[frameNum/4];

            else
                currentFrame = &kidMove[0];
            break;
    }

    kidSpriteSheet.SetColor( r, g, b );
    kidSpriteSheet.Render(posX,posY,currentFrame,angle,NULL,SDL_FLIP_NONE);
}

double Player::GetAngle()
{
    return angle;
}

SDL_Rect Player::GetRect()
{
    return collider;
}

void Player::SetState(playerstates state)
{
    this -> state = state;
}

playerstates Player::GetState()
{
    return state;
}

Zombie::Zombie()
{
    //Initialize the offsets
    posX = 500;
    posY = 500;

    //Set collision box dimension
    collider.w = ZOMBIE_WIDTH-40;
    collider.h = ZOMBIE_HEIGHT-40;
    collider.x = posX+20;
    collider.y = posY+20;

    //Initialize the velocity
    velX = 0;
    velY = 0;

    //Initialize the angle
    angle = 0;

    //Initialize Default State
    state = IN_PLAY;
}

void Zombie::HandleEvent( SDL_Event& e )
{
    //If a key was pressed
    if( e.type == SDL_KEYDOWN && e.key.repeat == 0 )
    {
        //Adjust the velocity
        switch( e.key.keysym.sym )
        {
            case SDLK_UP: velY -= ZOMBIE_VEL; angle = 180.0; break;
            case SDLK_DOWN: velY += ZOMBIE_VEL; angle = 0.0; break;
            case SDLK_LEFT: velX -= ZOMBIE_VEL; angle = 90.0; break;
            case SDLK_RIGHT: velX += ZOMBIE_VEL; angle = 270.0; break;
        }
    }
    //If a key was released
    else if( e.type == SDL_KEYUP && e.key.repeat == 0 )
    {
        //Adjust the velocity
        switch( e.key.keysym.sym )
        {
            case SDLK_UP: velY += ZOMBIE_VEL; break;
            case SDLK_DOWN: velY -= ZOMBIE_VEL; break;
            case SDLK_LEFT: velX += ZOMBIE_VEL; break;
            case SDLK_RIGHT: velX -= ZOMBIE_VEL; break;
        }
    }
}

void Zombie::Move( Tile* tiles[], Uint8* r, Uint8* g, Uint8* b )
{

    //Move the Zombie left or right
    posX += velX;
    collider.x += velX;

    //If the Zombie collided or went too far to the left or right
    if( ( posX < 0 ) || ( posX + ZOMBIE_WIDTH > SCREEN_WIDTH ))
    {
        //Move back
        posX -= velX;
        collider.x -= velX;
    }

    if(TouchesWall(collider, tiles) )
    {
        //Move back
        posX -= velX;
        collider.x -= velX;
        *g = 32;
        zombieHealth -= 25;
        strcpy(zombieHealthMessage, "Zombie Health: ");
        snprintf(zombieHealthStatus,5,"%d",zombieHealth);
        strcat(zombieHealthMessage,zombieHealthStatus);

        if(zombieHealth == 0)
        {
            this->SetState(DEAD);
        }
    }

    else
    {
        *g = 255;
    }

    //Move the Zombie up or down
    posY += velY;
    collider.y += velY;;

    //If the Zombie collided or went too far up or down
    if( ( posY < 0 ) || ( posY + ZOMBIE_HEIGHT > SCREEN_HEIGHT ) )
    {
        //Move back
        posY -= velY;
        collider.y -= velY;
    }

    if(TouchesWall(collider, tiles) )
    {
        //Move back
        posY -= velY;
        collider.y -= velY;
        *g = 32;
        zombieHealth -= 25;
        strcpy(zombieHealthMessage, "Zombie Health: ");
        snprintf(zombieHealthStatus,5,"%d",zombieHealth);
        strcat(zombieHealthMessage,zombieHealthStatus);
        if(zombieHealth == 0)
        {
            this->SetState(DEAD);
        }
    }
}

void Zombie::Animate( int frameNum, playerstates state, Uint8 r, Uint8 g, Uint8 b )
{
    //Animation Data
    SDL_Rect* currentFrame;

    switch (state)
    {
        case DEAD:
            currentFrame = &zombieDeath[0];
            break;
        case IN_PLAY:
            if(velX !=0 || velY !=0)
```

```
          currentFrame = &zombieMove[frameNum/4];

      else
          currentFrame = &zombieMove[0];
      break;
  default:
      if(velX !=0 || velY !=0)
          currentFrame = &zombieMove[frameNum/4];

      else
          currentFrame = &zombieMove[0];
      break;
}


zombieSpriteSheet.SetColor( r, g, b );

zombieSpriteSheet.Render(posX,posY,currentFrame,angle,NULL,SDL_FLIP_NONE);
```

6. Sample Outputs

Fig 6.1:Screenshot of Start Screen:

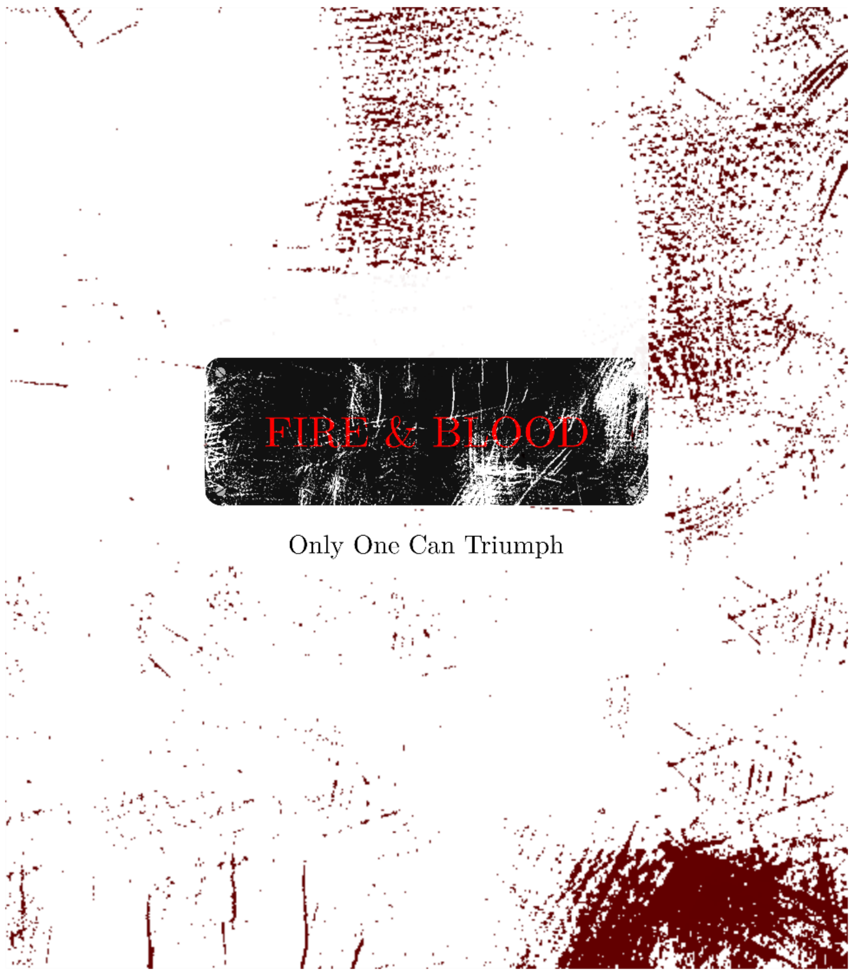
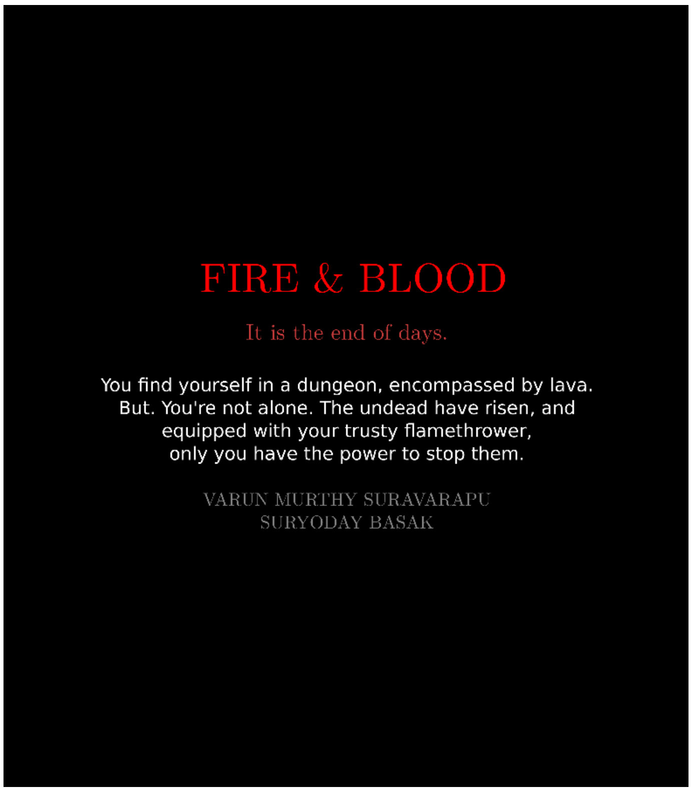


Fig 6.2:Screenshot of Description Screen:

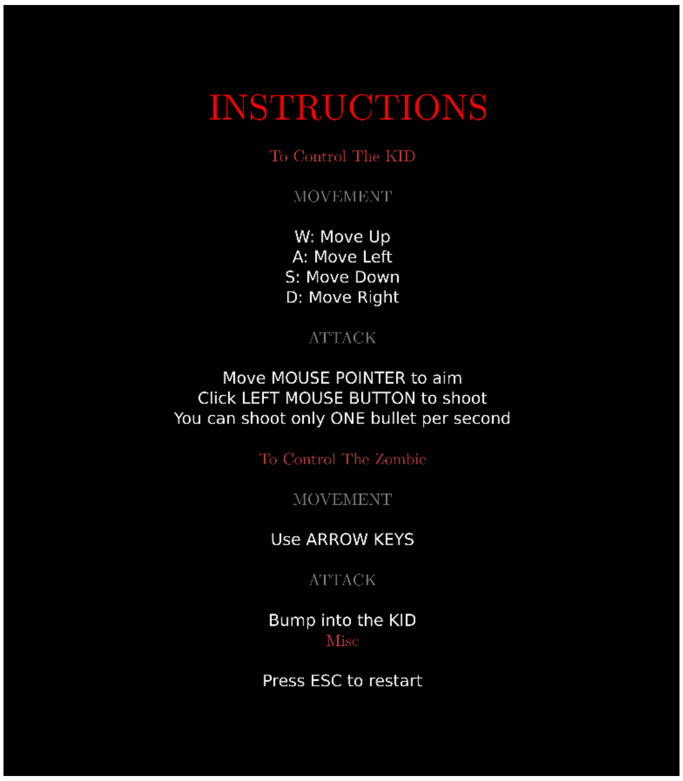

FIRE & BLOOD

It is the end of days.

You find yourself in a dungeon, encompassed by lava.
But. You're not alone. The undead have risen, and
equipped with your trusty flamethrower,
only you have the power to stop them.

VARUN MURTHY SURAVARAPU
SURYODAY BASAK

Fig 6.3:Screenshot of Instruction Screen:


INSTRUCTIONS

To Control The KID

MOVEMENT

W: Move Up
A: Move Left
S: Move Down
D: Move Right

ATTACK

Move MOUSE POINTER to aim
Click LEFT MOUSE BUTTON to shoot
You can shoot only ONE bullet per second

To Control The Zombie

MOVEMENT

Use ARROW KEYS

ATTACK

Bump into the KID
Misc

Press ESC to restart

Fig 6.4: Screenshot of In-Game Screen:

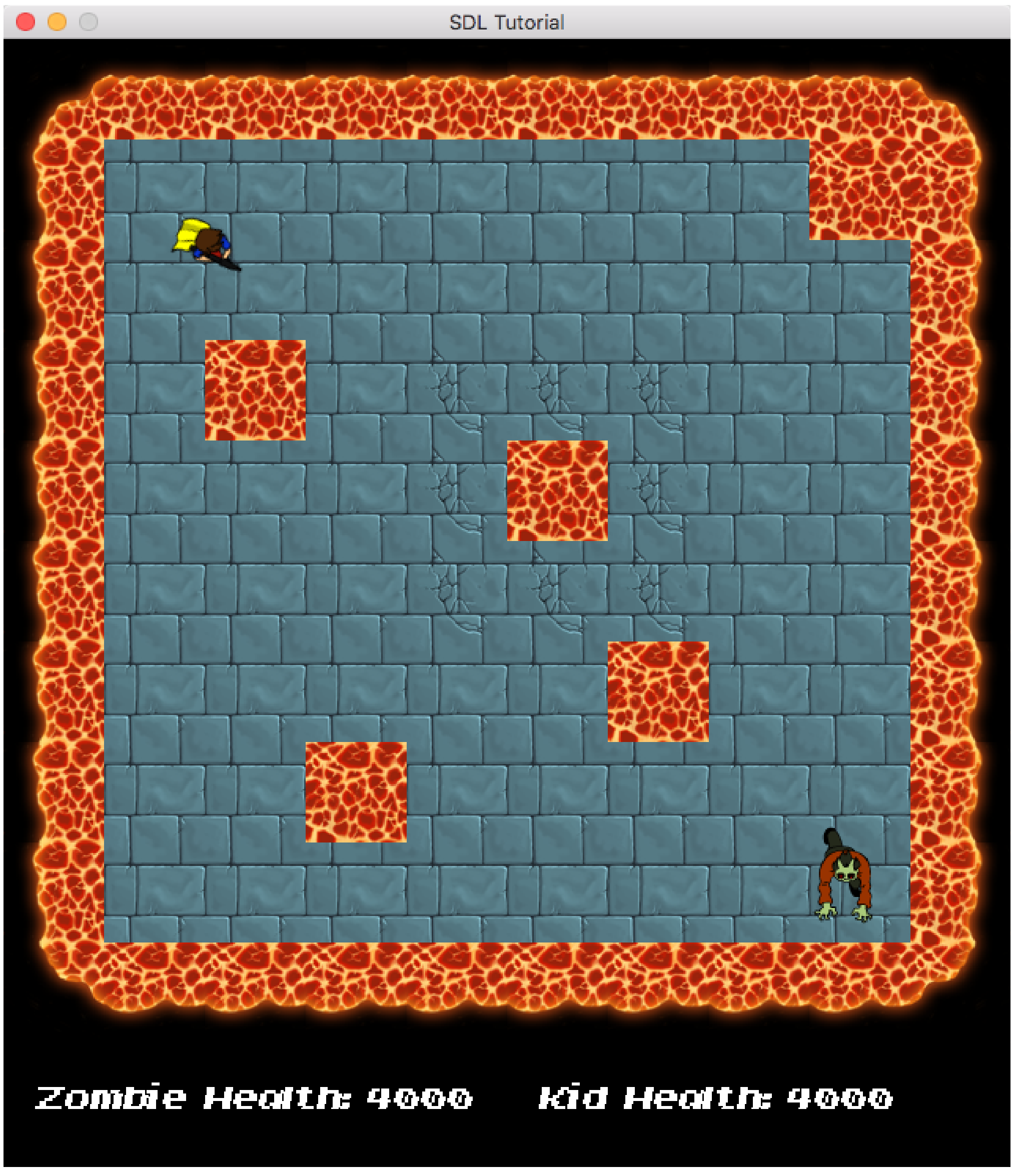
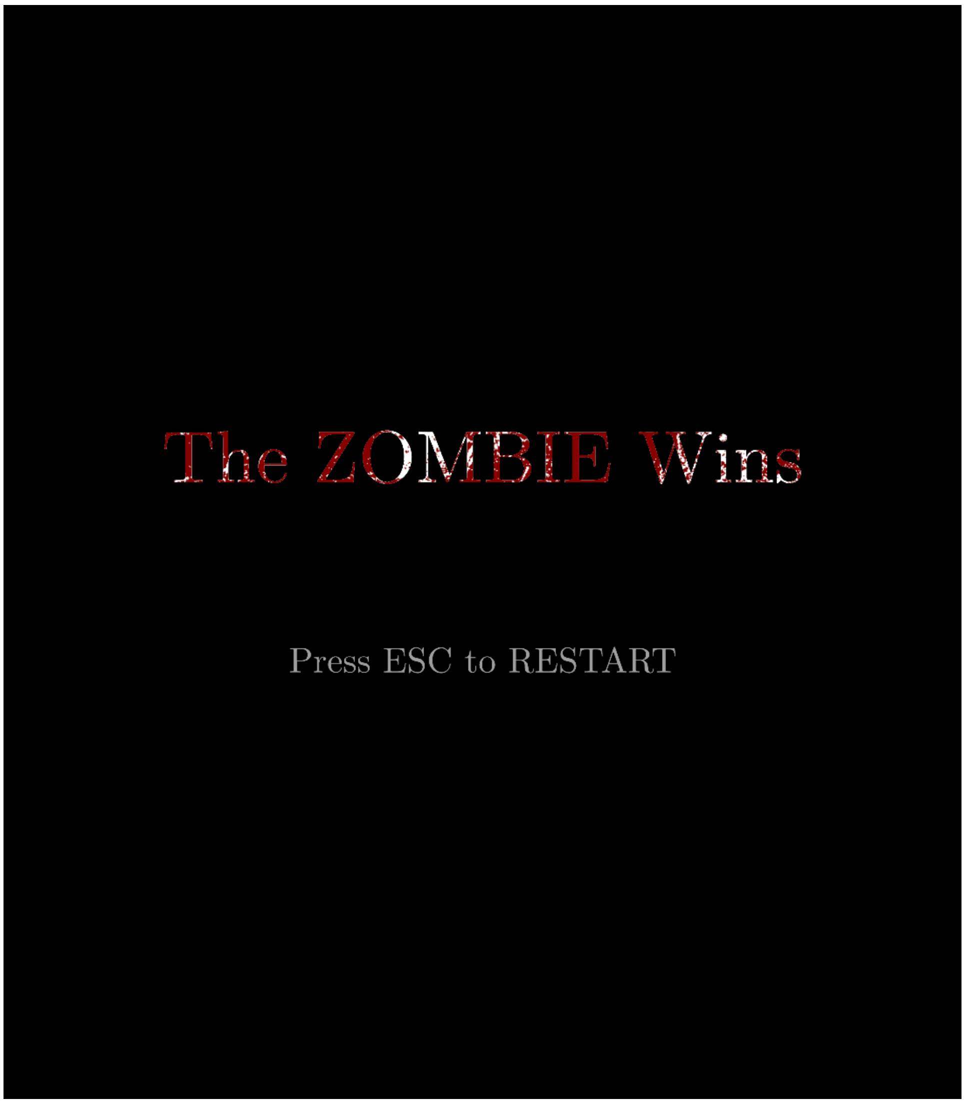


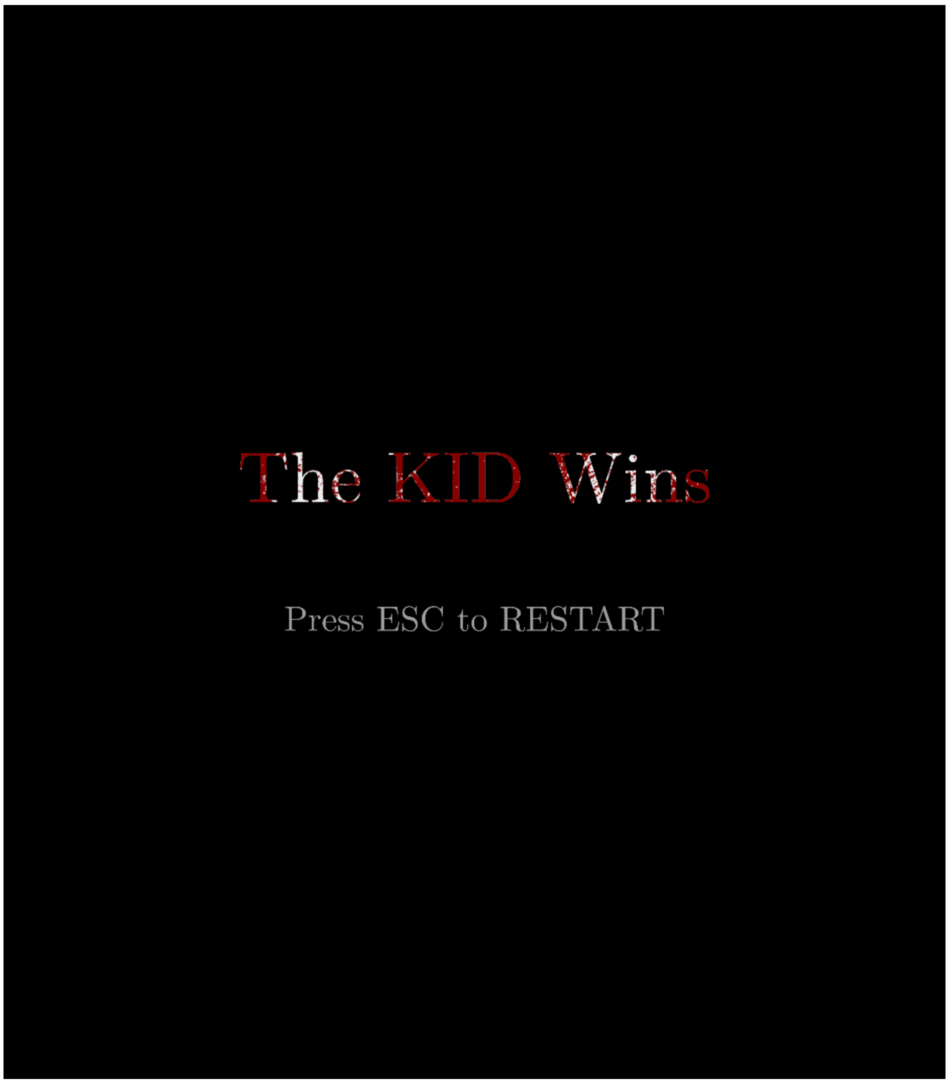
Fig 6.5: Screenshot of Zombie Win Screen:



The ZOMBIE Wins

Press ESC to RESTART

Fig 6.6: Screenshot of Kid Win Screen:



7. Conclusion

In the development of this project, we explored concepts of Computer Graphics taught to us as a part of our course in Computer Science and Engineering. In the process of development, we learnt to use the libraries The OpenGL Utility Toolkit (GLUT), and specifically for this project, the Simple DirectMedia Layer (SDL) in C++.

The game involves a top-down view of a two-player game. The protagonist of the game is a boy, who is controlled using the W-A-S-D style control of lateral movement, and mouse for pointing and shooting. The antagonist of the game is a zombie, who is controlled using the arrow key. The objective of the game is to reduce the opponent's health to zero; the boy can accomplish this by shooting at the zombie, and the zombie may win by bumping into the boy. The game's maps are dynamically generated based on a set of training maps which are fed prior to the start of the game instance.

SDL is compatible with the latest version of OpenGL, that is version 4, and makes the window functions, interactions based on keyboard and mouse, etc. more intuitive to handle. We also explored the dynamic generation of maps based on a set of training maps, which is introduces a dimension of Machine Learning to Game Development.

8. Future Scope

The project gives insights in the application of machine learning in computer games. Our project serves as a prototype for further work in generation of maps in games which may not be statically defined. The advantage of the approach is that the programmers need to explicitly define rules to dynamically generate terrain in a game. This is particularly useful in situations where maps do not need to be generated deterministically. For example, a multiplayer first person shooter game would be more challenging if the map which is generated each time is slightly different from the previous maps. It would make the game a lot more challenging as well.

9. Bibliography

Websites Referred

- http://www.opengl.org/ - For Finding OpenGL Library Information
- https://wiki.libsdl.org/ - SDL Documentation of it’s extensive API’s
- http://lazyfoo.net - For his Amazing SDL Tutorial Set
- http://stackoverflow.com - For Doubts and Queries on Coding and Implementation
- http://opengameart.org - For Royalty Free Sprites and Artwork
- http://texturepacker.com - To Create Spritesheets which enabled easy animation

Books Referred

- Donald Hearn and Pauline Baker. *Computer Graphics – OpenGL*
- F. S. Hill Jr. *Computer Graphics Using OpenGL*
- James D. Foley, Andres Van Dam, Steven K. Feiner, John F. Highes. *Computer Graphics*, Pearson Education