**INTRODUCTION**

* 1. **Background and Objectives**

As per the ENGG 102 course requirement of the first year second semester of Computer Engineering in Kathmandu University, we created a simple multiplayer arcade game called ‘The Tank Game’ using C programming language. This game is adapted from the very popular *Nintendo’s* console game “BATTLE CITY”. Original game includes single player and double player modes. The objective of the player is to protect their queen from the enemy tanks. We have implemented similar concept for our game with some major and minor modification. Modifications are discussed latter in the section.

There has been observed vacancy in interactive and simulated style of game development. We are generally familiar with the monotonous single player console based game that lack necessary degree of entertainment. User sooner or later gets hold of such games and ends up deleting such programs. Technology has greatly moved on from the time of simple boxed style gaming to today’s high-end virtual gaming. But still user finds simple 2D games interesting.

Our effort here is to develop a 2D multiplayer arcade game that consists of two or four player mode to increase excitement and provide user with some real fun. Besides the multiplayer aspect we have included nice graphics and sound effect to the game that generally lacks in the old console games. Our main motto is to provide all the thrill and excitement to the user. We believe we have achieved most of our goals through this project.

* 1. **Modification from Nintendo’s “BATTLE CITY”**

Nintendo’s “BATTLE CITY” came out in late 1980’s when there was a rapid growth in console based games. The game became an instant hit and is still very popular now. This game is very interesting to play. It consists of single and double player modes. Players are located centrally at a battle city. The main objective of players is to protect their Queen from the enemy tanks that are in fixed number and appear at certain intervals during the game.

We adapted similar concept for our game with some major and minor modification. The modifications in our game are listed in the table below.

|  |  |  |
| --- | --- | --- |
| Features | Nintendo’s “BATTLE CITY” | The Tank Game |
| Number of Players | Single or Double player mode | Double of Quadra duple mode |
| Enemies | Computer controlled tanks with simple AI | Computer Controlled enemies with no AI |
| No of Seals | One | Two |
| Objective of the game | To protects own seal from the enemy tanks | To protect own seal in the mean time destroy opponent’s seal |
| No of Levels | 35 | 13 |
| Mode of game play | Console based | Window based |

Table : Modification in the Tank Game

* 1. **Feasibility Analysis**

After having discussion with the probable users of the game “The Tank Game”, we came to a point that creating a game was feasible for them. We have researched on the current system’s working and possible resources available for the creation of the game.

A feasibility study is a preliminary study undertaken before the real work of a project starts to ascertain the likelihood of the project’s success. It is an analysis of possible alternative solutions to a problem and a recommendation on the best alternative. Within a feasibility study, three areas much be review, including Technical, Schedule, Operational.

* + 1. Technical Feasibility

Technical feasibility involves questions such as whether the resource needed for the development of the game exits, how difficult will be to build, and whether we have enough skills and experience needed for the development. We studied the technical feasibility of our project to identify the feasibility and concluded to use C programming language because of its efficiency and flexibility for the further extensions and compatibility.

Secondly due to limited support for the interactive gaming in C programming standard libraries we decided to use third party Game Development Library “Simple Direct Media Layer (SDL)” for our project.

Finally, it was necessary to develop good graphics, add exciting sound effects to our game to make the game play more fun. For that we chose Adobe Photoshop, Macromedia Flash, Microsoft Paint and Abssy Wave Editing software’s for graphics and sound editing.

1.3.2 Schedule Feasibility

This involves questions such as how much time is available to build the new game, when it can be built, etc. We made schedule feasibility to make ensure that we are able to complete project within specified time.

1.3.3 Operational Feasibility

In this feasibility, system, organizational and human aspects in order to ensure that the proposed 1`game will work when developed. We made the operational feasibility study to ensure that the software will operate with its flexibility and compatibility with new system and found that efficient use of the SDL library and flexible algorithms will enable to add new extensions in order to operate it efficiently in future.

* 1. **Software Requirements Specification**
     1. Software Requirements:
        + Microsoft Windows Operating System ( XP and higher)
     2. Hardware Requirement
        + Physical Memory(RAM) 512 MB minimum
        + Virtual memory double the amount of RAM
        + Processor 1.8 GHz minimum
        + Video adapter 256 colors
     3. Problem Specific Requirements:
        + SDL Library environment
  2. **System Overview**
     1. Graphics and Sounds

Graphics and sounds are essential part of any game. They account for the entertainment and excitement provided by the game. It was necessary to provide good graphics and sound support to our game. For this we implemented SDL library’s built in features that enabled us to load images and sounds from system to our game and use them accordingly. Detail implementation of the *SDL Image* and *SDL Mixer* is provided in the implementation section of the report.

* + 1. Multiplayer Gaming:

Multiplayer games are more popular than single player games; this is largely due to interaction between the users that make the game play more exciting. The main thrive of our project was to provide multiplayer gaming experience. Our game provides two players and four players’ mode. Single player mode is supported. Two players each from one team play from the keyboard and other two players play from the joysticks.

The keyboards and joysticks control map of each player are described below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| S.N. | Operation | Controls Keys | | | |
| Player1  (Keyboard) | Player2  (Keyboard) | Player3  (Joystick) | Player 4  (Joystick) |
| 1. | Move Up | UP key | W | Y-axis UP | Y-axis UP |
| 2. | Move Down | DOWN | S | Y-axis DOWN | Y-axis DOWN |
| 3. | Move Left | LEFT | A | X-axis LEFT | X-axis LEFT |
| 4. | Move Right | RIGHT | D | X-axis RIGHT | X-axis RIGHT |
| 5. | Fire Bullet | L | F | Button 1 | Button 1 |
| 6. | Pause Game | P | | | |
| 7. | Resume Game | R | | | |
| 8. | Pause/ Resume Sound | 1 | | | |

Table : Users control keys setting

* + 1. Event Handling

Any game involves handling of large number of events simultaneously. Events of the game may include motion of player, motion of enemy, firing bullets. In order to handle the events that are generated while playing our game we used different SDL library structure and functions that provide with necessary events handling. Table 3 describes all the events that occurs in the game and the how the event handling is achieved.

In case of event handling by implementation of SDL library, we are provided with an event structure that holds different events generated by the user. We use this event structure to determine what event has been generated by the user. Based on this, we carry out require task in the game.

|  |  |  |  |
| --- | --- | --- | --- |
| Events in the Game | Event handling methodology | Function and Structures used | |
| Function | Structure |
| Movements of the Player | Take inputs from the keyboard and Joysticks for the player. | Move\_player()  Check\_Collision()  Check\_Player\_Collision()  Collision() | Event  PLAYER |
| Firing bullet | Determine the visibility of the bullet and move the bullet if visible. If the bullet is not already visible than set the bullet visible. | Move\_Bullet()  Check\_Collision()  Collision() | Event  BULLET |
| Button click | Determine the coordinates of the buttons and coordinate of the mouse. Compare the coordinates and determine the mouse click. | Check\_Click() | BUTTON |

Table : Events generated by the user in the game and event handling.

* + 1. Score Management

It is necessary for any interactive game to provide an efficient score management system. Scores provide user with the options to compare their gaming skills with other players, this adds to the excitement of the game.

Score management in our game is implemented using simple file handling techniques. When a particular team score falls in our top test score list then they are prompted to enter their name. This information is then saved in the file “Scores.dat” located in the game directory. User can view top ten scores at any time from the main welcome screen by clicking on the “Hall of Fame” button.

* + 1. User Interactions

User interaction involves interaction of the user with the game. Success story of any game includes the features it provides for user interactions. When users are able to interact with the game they are lured towards it and prevents the game from being monotonous.

In order to implement effective user interaction with our game we provided following features in our game:

* Main Welcome screen: This is the main game screen that is displayed at first when the game is loaded. This screen includes 5 options. They are:

Play Game

Options

Hall of Game

Credits

Exit

These buttons enable user to explore the game. They can turn on and off sound in option menu. They can view top 10 score by click on “Hall of Fame” button. Finally users can view who are the developers of the game by clicking on the Credits button.

* Score Board: Score board is located at the right side of the game play screen. It maintains a track of scores of each team along with the remaining lives of each player. It also includes the current level.
* Game over Screen: This screen is displayed at the end of the game. It prompts the users to enter their name if they managed to get top 10 score else it just displays the high score along with users scores. It also provides users with option to play the game again.

**DISCUSSION**

* 1. **The C Language**

Different programming languages are used for programming games. Out of them C programming language is most often chosen by the programmers due to following features provided by it:

* It is easy use and understand
* It provides easy support for flexible and extensible style of programming.
* It has very limited restrictions when it comes to hardware’s access and utilizations. This enable user to directly manipulate devices for interactive gaming.
* Pointers are very powerful tools of C language that enable game programmers to control various events and actions during the game.
* Easy to use file handling system for good score management system.

But it has some limitations also:

* Lack of real world modeling of programming problem makes it difficult to implement big programs easily.
* Globalization of variables and unrestricted access to them makes the program hard to debug and test.
* Although pointer are strong hold of C, but incorrect use to them may lead to disastrous result.

By far it outweighs all other language when it comes to developing 3D games. Almost all high-end games are programmed in C today. Thus considering all these features we decided to use C programming language for our project.

* 1. **Resource Used**

Besides using C programming language for our game development we used other resource for various purposes in the game. They are described in details below:

* + 1. Simple Direct Media Layer(SDL)

Standard library that is provided by the C compiler manufacturer doesn’t provide enough support for good game development. The graphics and sound support are poor. And handling of user events becomes a difficult task.

SDL is a free open source game development library that is entirely coded in C programming language. It is written over Open Graphics Library (OpenGL), this help to simplify OpenGL providing easy functionality to handle its function and structures. SDL is a composite of many small libraries that can be incorporated together to develop games. Various SDL libraries implements in our game are listed below with their implementation in the game.

|  |  |  |
| --- | --- | --- |
| S.N. | SDL Library Component | Implementation in the game |
| 1. | SDL | It is the main library of the SDL package as is used for generating main screen, event handling, for SDL structures like SDL\_Rect, SDL\_Color, SDL\_Surface. Other implementation of this library is for images handling. |
| 2. | SDL\_Image | This Library provides special features to handle images in the game. They are importing images to the game using the *IMG\_Load()* function. |
| 3. | SDL\_Mixer | This Library provides support for sounds and music effect for the game. It has function like *Mix\_PlayMusic(), Mix\_HaltMusic()* and other functions that enables us to control sound play in the game. Besides it also provides with the *Mix\_Music* and *Mix\_Chunk* structures that are used for holding game music and sound effect. |
| 4. | SDL\_Font | This library provides us with the support for fonts and displaying messages in the game. It has TTF\_Font structure for loading fonts and functions like TTF\_RenderText\_Blender() for displaying the texts. |

Table : Componets of SDL Library used in the game

A lot of credits go to the SDL Library for successful implementation of the project. It is not hard to point out that we would not have accomplished the task with using this library. Links to more information and resources for SDL library is given in Glossary.

* + 1. Graphics Editing

It was necessary for us to provide our game with good graphics. The whole point of using graphics in a game is to make the game play more fun. We required the software by which could easily design and modify various graphics for our game. What better option we had then to use the best software available for the purpose, which is *Adobe Photoshop CS*. Adobe Photoshop CS is a high featured commercial photo editing software available in the market today.

In particular we used this software to deign graphics for our game objects. They include images for all the game components like players, enemies, bullets, walls, bricks etc. A List of all images used in the game is provided in the APPENDIX A. Addition of user generated graphics to our game was achieved by using SDL image library support.

Besides Adobe Photoshop CS we also used Microsoft Paint for designing graphics for the game.

* + 1. Animation

Our game includes various animations for different scenarios like the cracking and explosion of walls, explosion of players and seals. These animations were created in *Macromedia Flash MX* as sequence of bitmap images and were used like the flipbook pages in the game to generate a complete animations sequence.

Macromedia Flash MX has special support for creating animations clips and to export them as a sequence of bitmap images, one image for each frame. This features made the implementation of animations in our game successful.

* + 1. Sound Editing

Editing of sound and music clips was necessary to cut out noises and non-essential part of the music that intervened during the game play. In the mean time it was also necessary to convert all music and sound format to standard WAV and MP3 formats. These music formats are supported by the SDL Library. The purpose of sound editing and conversion was accomplished using *Abyss Wave Editor*.

Abyss Wave Editor Provides easy functionality to cut out unwanted parts from a music file and then save the final product into desired format.

* 1. **Project Implementation**
     1. Adding and manipulating Graphics

Graphics is the essential part of our game. We custom designed our graphics by using graphics editing software’s. The main challenge was to incorporate these graphics into our game. This was accomplished by using following SDL Library functions:

|  |
| --- |
| **Function Name:** IMG\_Load  **Function prototype:** SDL\_Surface \* IMG\_Load( const char \* file\_name);  **Number of Arguments:** 1  **Return Type:** SDL\_Surface \*  **Return Value:** Return a pointer to the image after loading the file into computer memory.  **Arguments Description:**  Argument 1: Locations of the image file to be loaded.  **Function’s Description:**  This is a SDL image library function that loads image from the specified file into computer memory. On successful loading of file it return a pointer pointing to the location in the memory where the image has been loaded. This pointer is then assigned to the image pointer variable in the game and then the image is manipulated using this pointer. If it fails to load the image it returns a NULL pointer.  **Sample Implementation:**  Player\_image = IMG\_Load(“images\\player\_imageUP.bmp”);  If(Player\_image == NULL) return error; |
| **Function Name:** SDL\_DisplayFormat()  **Function prototype:** SDL\_Surface \* SDL\_DisplayFromat(SDL\_Surface \*image);  **Number of Arguments:** 1  **Return Type:** SDL\_Surface \*  **Return Value:** Return a pointer to the image after optimization.  **Arguments Description:**  Argument 1: The pointer to the image that is to be optimized  **Function’s Description:**  This is a SDL image library function that optimizes a given image (already loaded into computer memory) to the game screen format. Basically it converts the pixel information of image to pixels of the game screen.  **Sample Implementation:**  SDL\_Surface \*analyze\_image(SDL\_Surface \*src)  {  //Temporary sdl surfaces  SDL\_Surface \*loaded\_image; //to hold the loaded image  SDL\_Surface \*optimized\_image; //to hold the optimized image    loaded\_image=src;    //optimize the image  optimized\_image= SDL\_DisplayFormat(loaded\_image);    //Free the temporary buffer  SDL\_FreeSurface(loaded\_image);    //return the optimized image  return optimized\_image;    } |
| **Function Name:** SDL\_SetColorKey  **Function prototype:** int SDL\_SetColorKey(SDL\_Surface \* image, SDL\_SRCCOLORKEY, SDL\_Color RGB);  **Number of Arguments:** 3  **Return Type:** int  **Return Value:** Return 0 if the conversion was successful else return -1 if there was an error.  **Arguments Description:**  Argument 1: Image pointer that is to be color keyed.  Argument 2: SDL flag that tells the function to make the pixels of given color transparent.  Argument 3: SDL color structure defining the color of the pixels that is to be made transparent specified with Red, Green and Blue values.  **Function’s Description:**  This is a SDL image library function that makes pixels of given color transparent and then return this modified image as a pointer. |
| **Function Name:** SDL\_FreeSurface  **Function prototype:** void SDL\_FreeSurface(SDL\_Surface \*image);  **Number of Arguments:** 1  **Return Type:** void  **Return Value:** Nothing  **Arguments Description:**  Argument 1: The image pointer pointing to the image that is to be freed from the computer memory  **Function’s Description:**  This is a SDL image library function that frees the resources used by a given image in the computer memory. This function is called at the end of the game when the images that have been previously loaded in the game are to be freed in order to free the resources used by the game and prevent Memory Leak.  **Sample Implementation:**  void unload\_images()  {  SDL\_FreeSurface(background);  SDL\_FreeSurface(wall\_image);  SDL\_FreeSurface(brick\_image);  SDL\_FreeSurface(bullet\_image);  SDL\_FreeSurface(explode);  SDL\_FreeSurface(seal\_image[0]);  SDL\_FreeSurface(seal\_image[1]);  SDL\_FreeSurface(seal\_image[2]);  SDL\_FreeSurface(cracking);    gameLog("All images were sucessfully unloaded...");  } |

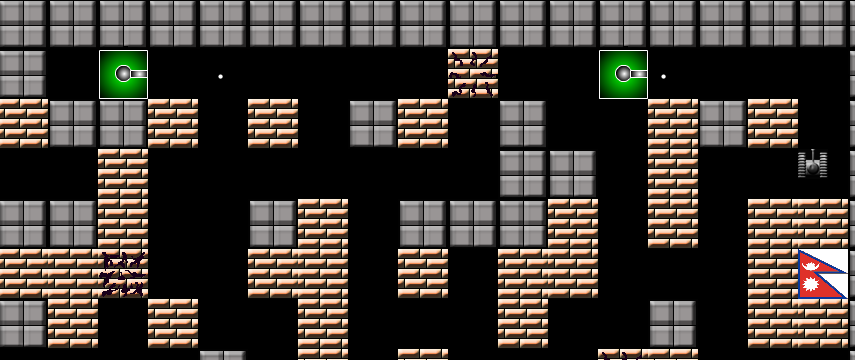
Here is a snapshot from the game screen showing various images that has been loaded in the game.

Figure : Graphics loaded in the game

* + 1. Adding Sounds

Sound effects were required for various events in the game. The Main background music plays with the start of the game. Other small sound effect like explosion sound of wall, player explosion, seal explosion, bullet fire sounds all were implemented in the game by using the SDL Mixer library. This library provides support for loading, playing and manipulating sounds as per requirement. Adding of sound by using the SDL mixer library was done using following functions:

|  |
| --- |
| **Function Name:** Mix\_LoadMUS  **Function prototype:** Mix\_Music \* Mix\_LoadMUS( const char \* file\_name);  **Number of Arguments:** 1  **Return Type:** Mix\_LoadMUS  **Return Value:** Return a pointer to the music after loading the file into computer memory.  **Arguments Description:**  Argument 1: Locations of the music file to be loaded.  **Function’s Description:**  This is a SDL mixer library function that loads music from the specified file into computer memory. On successful loading of file it return a pointer pointing to the location in the memory where the music has been loaded. This pointer is then assigned to the music pointer variable in the game and then the music is manipulated using this pointer. If it fails to load the music it returns a NULL pointer.  **Sample Implementation:**  int load\_sound()  {  background\_music = Mix\_LoadMUS("sounds\\music.wav");  click\_sound = Mix\_LoadWAV("sounds\\click.wav");  wall\_explode = Mix\_LoadWAV("sounds\\wall\_explode.wav");      if( background\_music == NULL ||  click\_sound == NULL ||  wall\_explode == NULL  ) return -1;    gameLog("All the sounds were sucessfully loaded...");  return 0;  } |
| **Function Name:** Mix\_PlayMusic()  **Function prototype:** int Mix\_PlayMusic(Mix\_Music \*music, int repetitions);  **Number of Arguments:** 2  **Return Type:** int  **Return Value:** Returns 0 on success else returns -1 indicating error in music play.  **Arguments Description:**  Argument 1: Pointer to the music file that is to be played  Argument 2: Number of times to play the music, -1 means to repeat the music for indefinite period.  **Function’s Description:**  This is a SDL mixer library function that allows us to play the given music file for desired number of times.  **Sample Implementation:**  Mix\_PlayMusic(background\_music,-1); |
| **Function Name:** Mix\_LoadWAV  **Function prototype:** Mix\_Chunck \* Mix\_LoadWAV(char \*file\_name);  **Number of Arguments:** 1  **Return Type:** Mix\_Chunk\*  **Return Value:** Return the pointer location to the WAV file loaded in the memory.  **Arguments Description:**  Argument 1: Location of the WAV music file.  **Function’s Description:**  This is a SDL Mixer library function that allows us to load small music effect require for the game. |
| **Function Name:** Mix\_FreeMusic  **Function prototype:** void Mix\_FreeMusic(Mix\_Music \*music);  **Number of Arguments:** 1  **Return Type:** void  **Return Value:** Nothing  **Arguments Description:**  Argument 1: The music pointer pointing to the location where the music file is loaded.  **Function’s Description:**  This is a SDL image library function that frees the resources used by a given music in the computer memory. This function is called at the end of the game when the music that have been previously loaded in the game are to be freed in order to free the resources used by the game and prevent Memory Leak.  **Sample Implementation:**  void unload\_sound()  {  Mix\_FreeMusic(background\_music);  Mix\_FreeChunk(click\_sound);  Mix\_FreeChunk(wall\_explode);  gameLog("All the sounds were unloaded...");    } |

* + 1. Level Management

Our game consists of various levels of game play. Each level has different arrangements for wall, bricks and enemies but the players and seals locations are always fixed. It creates an excitement and provides more fun to the users. Concept of level management was then required to successfully implement this feature in our game. This was implemented by using following functions:

|  |
| --- |
| **Function Name:** New Level  **Function prototype:** void new\_level(int level);  **Number of Arguments:** 1  **Return Type:** void  **Return Value:** Nothing  **Arguments Description:**  Argument 1: Level number to be loaded.  **Function’s Description:**  This function was designed to load a particular level. The level which is to be loaded is passed as an argument to the function. Function then call all necessary function require to load that particular level.  **Algorithm:**  Here is the algorithm required to carry out the task   1. First, Check whether the levels are complete 2. Clean all cell structures 3. Display the level up screen 4. Read the level information from the level information file 5. Draw out the entire level 6. Update the scoreboard 7. Record the progress in the game log 8. Finally if the levels are complete then display the game over screen.   **Sample Implementation:**  void new\_level(int level)  {  //Check if the total numbers of levels provided by the game  //is complete  if(level<TOTAL\_LEVELS)  {    //clear our all cell structures  clean\_cells();  //display the level up screen  level\_up\_screen(level);    //read the level information from the file  read\_level\_file(level);    //draw out the entire level  draw\_level();    //load all necessary objects of the level.  load\_object();    //update the score board  load\_scoreboard();    //record in the log  fprintf(game\_log,"\n NEW LEVEL INITIALIZED : %d",level);  }  else //if all levels are complete  {  //call the game over screen to carry out the proceedings.  game\_over();  }  } |
| **Function Name:** level\_up\_screen()  **Function prototype:** void level\_up\_screen(int level);  **Number of Arguments:** 1  **Return Type:** void  **Return Value:** Nothing  **Arguments Description:**  Argument 1: Current level number.  **Function’s Description:**  This function calls the level up screen to be displayed. Level up screen display the new level that is being loaded and a short message to excite user interest. It hold for 3 secs.  **Algorithm:**  The algorithm for the level up screen function is   1. Clear out the previous screen with a black image 2. The prints the current level number 3. Print the special message 4. Hold the screen for 3 seconds   **Sample Implementation(pseudo code):**  Void level\_up\_screen(int level\_no)  {  Clear the previous screen  Print “Level” + level\_no  Print “special message”  Delay (3 seconds)  } |
| **Function Name:** Read Level File  **Function prototype:** int read\_level\_file(int level);  **Number of Arguments:** 1  **Return Type:** int  **Return Value:** Return 0 if there was no error is reading the file information else returns -1 indicating error.  **Arguments Description:**  Argument 1: Level file number to be loaded.  **Function’s Description:**  This function reads the level information for a particular level from the level information file in the “levels” directory of the game directory. This information is then loaded in the cell structures of the game. It is then used for controlling a particular level. Information in the file are stored as number codes for each game elements. Cells are assigned each object based on the number codes read from the file.  **Algorithm:**  This function uses a simple algorithm to read the level information from the file:   1. Open the file from the level directory for the given level 2. Iterate a loop to read information for each cells 3. Check each elements with the number code and then assign the cell its respective object 4. Close the file   Here is the pictorial representation of the method in which a particular level information is stored in the level information file.  level_file.png  Figure : Method in which game object are numerically represented in a level information file  **Sample Implementation(pseudo code):**  Void read\_level\_file(int level)  {  File = openfile( level.txt)  For(i=0 to total number of cells)  {  Switch(read(i))  {  If wall, than cell[i] = wall  If brick, than cell[i] =brick  }    }  } |
| **Function Name:** Draw Level  **Function prototype:** void draw\_level();  **Number of Arguments:** 0  **Return Type:** void  **Return Value:** Nothing  **Arguments Description:**  None  **Function’s Description:**  This function does the actual task of drawing object of the game to the screen. It uses *draw\_image()* function to draw the image. In the function we have iterated from beginning to end of the cell counts, checking what is present in each cell. Based on the cell content we have drawn the images of the object in the screen. This function must be called only when the information of the level has been read from the file, because it depends on the information assigned to each cell by the real\_level\_file() function.  **Algorithm:**  The algorithm designed for this function is simple, it is given below:   1. Iterate a loop from the start of the cell to the end of the cells 2. Check what is present in each cell 3. Draw the image of the object at the location of the cell based on the information that is present in the cell 4. Refresh the screen to finally make the images visible   **Sample Implementation(pseudo code):**  void draw\_level()  {  for i = 0 to TOTAL\_CELL\_COUNT  {  switch(cell.isWhat)  {  case WALL:  Draw wall;  case BRICK:  Draw Bricks  case ENEMY:  Draw Enemy  }  }  Refresh(screen)  }  Here is the sample shot of a completely drawn level screen:  screen_shot1_draw_level.png  Figure : A Completely drawn level using draw\_level() function |
| **Function Name:** Load Objects  **Function prototype:** void load\_objects();  **Number of Arguments:** 0  **Return Type:** void  **Return Value:** Nothing  **Arguments Description:**  None  **Function’s Description:**  In each level of the game each players begins to play from their default. So order to move player from their location in previous level to their default location in new level this function assigns default coordinates for each player locations.  Besides each bullets in the previous level are randomly located so they need to get their default location in this new level.  The main purpose of this function is to assign each of the game elements its defaults value for each level before the players begins to play the level.  **Algorithm:**  Algorithm behind the load\_objects() function is given below:   1. Iterate a loop for all the player in the game 2. Assign each player of the game their default coordinates and direction 3. Iterate a loop for all the bullets of the game 4. Assign coordinates and default location for each bullets 5. Iterate a loop for all the enemies in the level 6. Assign enemies their location and a random direction   **Sample Implementation(pseudo code):**  void load\_object()  {  for( i =0 to TOTAL\_PLAYERS\_COUNT)  {  player[i].x = default X;  player[i].y = default Y;  player[i].visibiliy = true  }  for(i = 0 to TOTAL\_BULLETS\_COUNTS)  {  bullet[i].x = default X;  bullet[i].y = default Y;  bullet[i].visibility = false;  }  for(i=0 to TOTAL\_ENEMY\_COUNT)  {  enemy[i].x =default X;  enemy[i].y = default Y;  enemy[i].direction = random();  }    } |

* + 1. Objects motions and control

Players, enemies and bullets are the motile objects of the game. Player is moved through keyboard and joysticks. Enemy is computer operated and bullets are operated by computer based on their visibility and direction of travel.

An operation of players involves moving the player in desired direction based on the user input. During the operation of player necessary function are used to check the collision of player with other object of the game. Also when the user presses the fire button the bullet for the player is activated in the given direction of the player.

Enemy on the other hand is computer controlled. Enemy rotates anti-clockwise firing bullet in each direction. This operation of enemy is controlled by implementation of appropriate function in the game.

All these motions and controls are implemented in the game using following functions:

|  |
| --- |
| **Function Name:** Move Player  **Function prototype:** int move\_player(Player \*player\_name, int direction);  **Number of Arguments:** 2  **Return Type:** int  **Return Value:** return 0 if success else 1 for error  **Arguments Description:**  Argument 1: A pointer to the player structure variable that is to be moved  Argument 2: Numerical value indicating the direction in which the player is to be moved.  **Function’s Description:**  This function moves the given player in given direction. But before it goes on to moving the player in the specified direction it makes necessary function calls to check if the player object makes a collision with other object. If the collision is made than the whole operation of moving the object is aborted. If no collision occurs in moving the player in the direction than it is moved in the given direction with given player speed.  The function used to check collision of player with other objects of the game are describes in the collision detection under implementation section.  **Algorithm:**  The necessary algorithm behind implementation of this function is:   1. Get the player Id that is being moved. 2. Update the image of the player with the image that suits the given direction of motion. 3. Check Collision of player with the other players in the game 4. Check the collision of player with the wall, bricks and seal i.e other cells components 5. Finally it no collision is made than move the player   **Actual Implementation:**  int move\_player(PLAYER \*pplayer,int Direction)  {  //For looping  int i=0;    //Determine the player id  int idd = pplayer->id;    //Clear out player previous image  clear\_image(pplayer->dimension);  //assign new direction to the player  pplayer->Direction=Direction;    //assign new image for the given new direction of the player  player\_image[idd][0] = player\_image[idd][Direction];    //This loops checks the collision of given player with all other REMAINING //players of the game except itself    for(i=1;i<5;i++)  {  if(i != idd)  {  if(player[i].isActive)  if(check\_player\_collision(player[idd], player[i],Direction) == true)  return 0;  }  }    //This parts checks the collision of player with wall or bricks and if no //collsion is detected than it moves the player in the direction requested  if(check\_collision(pplayer->dimension,Direction).isWhat == EMPTY)  {  //verify the direction of movement and move accordingly  if(Direction==UP) pplayer->dimension.y -= PLAYER\_SPEED;  if(Direction==DOWN) pplayer->dimension.y += PLAYER\_SPEED;  if(Direction==LEFT) pplayer->dimension.x -= PLAYER\_SPEED;  if(Direction==RIGHT) pplayer->dimension.x += PLAYER\_SPEED;  }  return 0;    } |
| **Function Name:** Move Bullet  **Function prototype:** int move\_bullet(BULLET \*bullet);  **Number of Arguments:** 1  **Return Type:** int  **Return Value:** Returns 0 on success else returns -1 indicating error.  **Arguments Description:**  Argument 1: Pointer to the bullet structure variable that is to be moved.  **Function’s Description:**  This function is concerned with the task of moving the bullet. Based on the visibility criteria of the bullet we call this function. This function moves the bullet in the direction that is pre-assigned to it during its activation. Function first moves the bullet than makes necessary function call to check whether the bullet has made collision with other object or not. Then it checks what the bullet has made collision with and performs the necessary action. And finally deactivates the bullet.  **Algorithm:**  The algorithm for the move\_bullet function is:   1. Get the bullet id and the id of the team by which it is fired. 2. Move the bullet in the given direction and update bullet locations 3. Check the collision of bullet with the cells elements bricks, wall, seals and enemies 4. Check the collision of bullet with the players 5. Check the collision of bullet with other bullets from players and enemies 6. If bullet makes any collision with the object then deactivate the bullet and perform necessary operations of the object that bullet collided with. |
| **Function Name:** Operate Enemy  **Function prototype:** int operate\_enemy(ENEMYY \*enemy);  **Number of Arguments:** 1  **Return Type:** int  **Return Value:** Return 0 if the operation was success else 1 if there was an error  **Arguments Description:**  Argument 1: The pointer to the enemy structure variable that is to be moved.  **Function’s Description:**  This function rotates the enemy in anti-clock wise direction firing one bullet in each direction. It also call the move\_enemy\_bullet() function to move the enemy bullet in the specified direction.  **Algorithm:**  The algorithm to operate the enemy is given below:   1. Check to see if the enemy bullet is already moving or not 2. If the bullet is not moving than rotate the enemy anti clock wise. 3. Assign the bullet new direction of the enemy. 4. Activate the bullet in the given direction 5. Finally draw the image for new enemy location.   **Actual Implementation:**  int operate\_enemy(ENEMYY \*eenemy)  {  //check is the enemy bullet is active or not  if(eenemy->my\_bullet.isActive == true)  {  //if active then move the enemy bullet  move\_enemy\_bullet(&(eenemy->my\_bullet),eenemy->id);  }  else  {  //if not active than rotate the enemy anticlock wise  eenemy->direction--;  if(eenemy->direction < 0 )  {  //if rotation is complete than reset the location of enemy  eenemy->direction = 4;  }  //assign the enemy bullet the direction the enemy  eenemy->my\_bullet.Direction = eenemy->direction;    //align the enemy bullet according the the direction //assigned to it  switch(eenemy->my\_bullet.Direction)  {  case ENEMY\_UP:  eenemy->my\_bullet.dimension.x = eenemy->dimension.x + 25;  eenemy->my\_bullet.dimension.y = eenemy->dimension.y;  break;  case ENEMY\_DOWN:  eenemy->my\_bullet.dimension.x = eenemy->dimension.x + 25;  eenemy->my\_bullet.dimension.y = eenemy->dimension.y + 15;  break;  case ENEMY\_LEFT:  eenemy->my\_bullet.dimension.x = eenemy->dimension.x;  eenemy->my\_bullet.dimension.y = eenemy->dimension.y + 25;  break;  case ENEMY\_RIGHT:  eenemy->my\_bullet.dimension.x = eenemy->dimension.x + 15;  eenemy->my\_bullet.dimension.y = eenemy->dimension.y+ 25;  break;    }    //set the bullet on motion  eenemy->my\_bullet.isActive = true;    }    //finally draw the enemy image  draw\_image(enemy\_image[eenemy->direction],screen,NULL,eenemy->dimension.x,eenemy->dimension.y);  } |
| **Function Name:** Move Enemy Bullet  **Function prototype:** int move\_enemy\_bullet(BULLET \*bullet, int enemy\_id)  **Number of Arguments:** 2  **Return Type:** int  **Return Value:** Returns 0 on success and 1 for error  **Arguments Description:**  Argument 1: Pointer to the enemy bullet structure that is to be moved  Argument 2: Id of the enemy firing the given bullet.  **Function’s Description:**  This function is similar to the *move\_bullet()* function for moving the player bullets. It is specially implemented for moving the enemy bullets that behave somewhat differently from the player bullets. So as to say, enemy bullet doesn’t kill other enemy neither there is collision between the enemy-enemy bullet.  **Algorithm:**  The algorithm for the move\_bullet function is:   1. Get the direction of motion of the bullet 2. Move the bullet in the given direction and update bullet locations 3. Check the collision of bullet with the cells elements bricks, wall, seals and enemies 4. Check the collision of bullet with the players 5. Check the collision of bullet with other bullets from players and enemies 6. If bullet makes any collision with the object then deactivate the bullet and perform necessary operations of the object that bullet collided with. |

* + 1. Collision detection

Collision detection is the crucial part of any game. Main objective of a game programmer is to determine the collision between various game elements effectively and perform the desired action.

Collision detection proved to be a big challenge for us initially. But this riddle was efficiently solved using following implementations:

|  |
| --- |
| **Function Name:** Collision  **Function prototype:** int Collision(SDL\_Rect ObjA, SDL\_Rect ObjB);  **Number of Arguments:** 2  **Return Type:** int  **Return Value:** Return 1(true) if the collision between the objects A and Object B is detected else return 0(false).  **Arguments Description:**  Argument 1: A SDL\_Rect structure variable holding the coordinates and dimension of the first object  Argument 2: A SDL\_Rect structure variable holding the coordinates and dimension of the second object  **Function’s Description:**  This is the simplest type of collision detection function that we have used in the game. The main purpose of this function if to detect collision between any two given object of the game.  The objects whose collision is to be detected are passed as *SDL\_Rect* structure variable that holds the coordinates of the object locations and dimensions of the objects.  Based on these information and using a simple algorithm for collision detection the collision between the objects is detected and equivalent value is returned  **Algorithm:**  The necessary algorithm behind implementation of this function is:   1. Determine the offsets of each object top,down, left, right. 2. Compare to see if the left of first object is greater than right of the second object in that case return false. 3. Compare to see if the right of the first object is less than left of the second object in that case return false. 4. Compare to see if the top of first object is greater than bottom of second object in that case return false. 5. Compare to see if the bottom of first object is less than the top of second object in that case return false. 6. Finally is all above condition are not met than return true(collision detected).   Here is the graphical representation of above algorithm.  **collision_method.png**  Figure :Collsion detection method using Collision() function  **Actual Implementation:**  int collision(SDL\_Rect objA, SDL\_Rect objB)  {    //dimension variables of the two objects colliding  int leftA,rightA,topA,bottomA;  int leftB,rightB,topB,bottomB;    //initialize dimensions variables  leftA= objA.x;  rightA =objA.x + objA.w;  topA = objA.y;  bottomA = objA.y + objA.h;    leftB= objB.x;  rightB = objB.x + objB.w;  topB = objB.y;  bottomB = objB.y + objB.h;    //see if the object A is above object return no collision if true  if(bottomA < topB ) return false;    //see if the object A is below object B return no collision if true  if(topA > bottomB) return false;    //see fi the object A is to the left of object B return no collision if true  if(rightA < leftB) return false;    //see if the object A is to the right of object B return no collsion if true  if(leftA > rightB) return false;    //finally if all conditions are met return true ie the collision has occured    return true;    } |
| **Function Name:** Check Collision  **Function prototype:** CELL check\_collision(SDL\_Rect obj, int direction);  **Number of Arguments:** 2  **Return Type:** CELL  **Return Value:** Returns the cell with its content that is located one step ahead of the given object in the specified direction.  **Arguments Description:**  Argument 1: SDL\_Rect structure variable containing the coordinates and dimensions of the object whose collision is to be detected.  Argument 2: Numerical value indicating the direction is which the collision is to be detected.  **Function’s Description:**  This function actually doesn’t check the collision for the object in the given direction, it just return what is located in the cell that is just in front of the given object in given direction. The collision between this returned object and the given object is then check using the Collision() function later in the game.  The implementation of the function is done by determining in which row and column the given player is located by using the coordinated of the object. By using this information we determine the cell that is located just ahead of the object in the given direction. Finally this detected cell with its content is returned by the function.  This function is used to detect the collision between the moving and stationery objects of the game only.  **Algorithm:**  The algorithm for the check\_collision() function is:   1. Get the offsets of the object 2. Determine the row and column number in which the player is located 3. Based on the direction of the movement determine the next row and column value 4. Using this row and column value access the cell grid structure to determine the exact cell that is located ahead of the object. 5. Return this cell with its content. |
| **Function Name:** Check Player Collision  **Function prototype:** int check\_player\_collision(PLAYER p1, PLAYER p2, int direction);  **Number of Arguments:** 3  **Return Type:** int  **Return Value:** Returns 1(true) if the collision between the players is detected else 0(false, no collision).  **Arguments Description:**  Argument 1: Player structure variable for the first player  Argument 2:Player Structure variable for the second player  Argument 3: Numerical representation of the direction in which the collision between the players is to be detected.  **Function’s Description:**  This function is specially drafted to detect collision between any two moving players in the game in the specified direction. This function eventually calls the *Collision()* function to detect the actual collision between the player, modification is that this function check collision between the players in the specified direction only. *Collision()* function on the other hand detects the collision between the objects in all direction. Check the collision of players in all direction renders player immotile. So we had to avoid it.  **Algorithm:**  The algorithm to operate the enemy is given below:   1. Increase the coordinates of the first player in the given direction temporarily. 2. Now check the collision between the players using the *Collision()* function. 3. Return true if the collision is detected else return false.   **Actual Implementation:**  int check\_player\_collision(PLAYER p1, PLAYER p2, int direction)  {  //This switch case incease the player coordinates based on the direction of movement of the  //player  switch(direction)  {  case UP:  p1.dimension.y -=PLAYER\_SPEED;  break;  case DOWN:  p1.dimension.y +=PLAYER\_SPEED;  break;  case LEFT:  p1.dimension.x -=PLAYER\_SPEED;  break;  case RIGHT:  p1.dimension.x +=PLAYER\_SPEED;  break;  }    //This parts check the colliison between the two player and return //true if the collsion is detected    if(collision(p1.dimension,p2.dimension) == true) return true;    //else return false indicating no collsion detected  else return false;  }    //set the bullet on motion  eenemy->my\_bullet.isActive = true;    }    //finally draw the enemy image  draw\_image(enemy\_image[eenemy->direction],screen,NULL,eenemy->dimension.x,eenemy->dimension.y);  } |

These three collision detection function works out the complete collision detection mechanism in our game. Following figure illustrates the relation between various collision detection function used in the game.

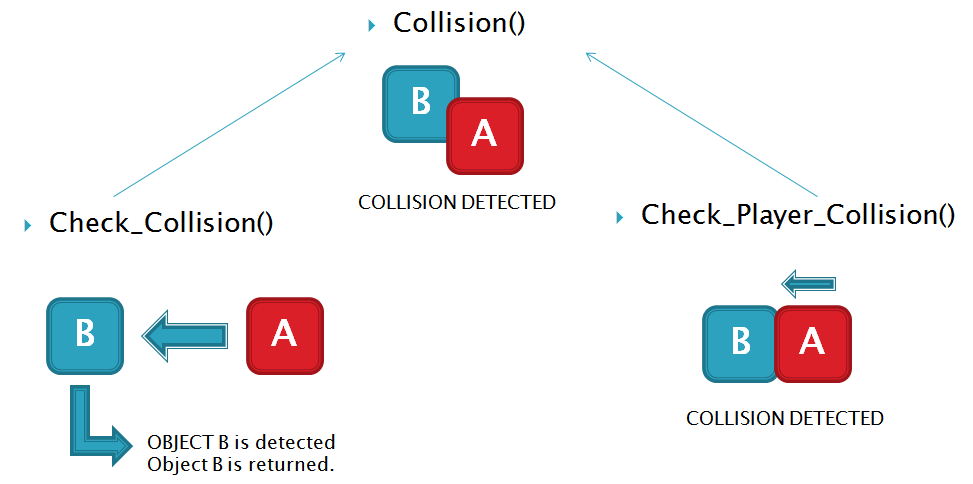


Figure 5 :Relation between 3 collision detection functions used in the game

* + 1. Event Handling

Main objective of any game is to provide fun and excitement to the users. Its accomplishes this task by enabling user to get directly involve with the elements of the game. Involvements includes moving object according to one wish.

In more technical term, event handling is the process of assessing events generated by the use during the game play. User generated events in the game includes key presses, button clicks, window events (close button click), joysticks motions etc. These events need to be handled in the game for enabling user interaction with the game. One cannot play the game by just watching it, they need to be involved by generating events.

Event handling in our game “The Tank Game” has been achieved using event handling feature of the SDL library. SDL includes an Event structure. All the user events generated during the game play is arranged by SDL in a queue, first in first out. We then use SDL\_PollEvent() function to poll the next event in the queue and SDL assigns it to our event structure. Then by manipulating this event structure handle the user generated events.

Detail implementation of the event handling is given below:

|  |
| --- |
| **Function Name:** SDL\_PollEvent  **Function prototype:** int SDL\_PollEvent(SDL\_Event \*event);  **Number of Arguments:** 1  **Return Type:** int  **Return Value:** Returns 1 if there is any pending event else return 0  **Arguments Description:**  Argument 1: A pointer to SDL\_Event structure variable which is assigned the event information by the function  **Function’s Description:**  This is a SDL Library function that is used to poll any pending user generated event. If any event is left than the specific information of the event is assigned to the event structure. This structure can then be check for what event has been generated and then manipulate the information accordingly.  **Algorithm:**  (SDL Library Function)  **Sample Implementation:**  SDL\_Event event; /\* Event structure \*/  .  .  .  /\* Check for events \*/  //Loop until there are no events left on the queue  while(SDL\_PollEvent(&event)){  switch(event.type){ /\* Process the appropiate event type \*/  case SDL\_KEYDOWN: /\* Handle a KEYDOWN event \*/  printf("Oh! Key press\n");  break;  case SDL\_MOUSEMOTION:  .  .  .  default: /\* Report an unhandled event \*/  printf("I don't know what this event is!\n");  }  } |
| **Function Name:** Check\_Click  **Function prototype:** int check\_click(BUTTON button, int mX, int mY);  **Number of Arguments:** 3  **Return Type:** int  **Return Value:** Return 1(true) if the button was click else returns 0(false).  **Arguments Description:**  Argument 1: A BUTTON structure variable that holds the offsets for the given button.  Argument 2: X coordinates of the location where the mouse was clicked.  Argument 3: Y coordinates of the location where the mouse was clicked.  **Function’s Description:**  This function was drafted to enhance user interaction with the game by using mouse click. It is specially used to check clicks for the button in main welcome screen, hall of fame screen, option screen, and the game over screen.  **Algorithm:**  Algorithm for the function is:   1. Compare is the x coordinate of the mouse click lies in between the x coordinate and (x + width) of the button. 2. If true, then compare if the y-coordinate of the mouse click lies in between the y-coordinate and (y + height) of the button 3. If true than return collision detected (1) else return collision not detected (0).   buttons.png  Figure : Button click in the main welcome screen, check\_click() function is invoked.  **Actual Implementation:**  int check\_click(BUTTON b, int x , int y)  {  if(x<(b.dimension.x+b.dimension.w) && x>b.dimension.x)  {  if(y>b.dimension.y && y<(b.dimension.y+b.dimension.h))  {  return true;  }  }  return false;  } |
|  |

* + 1. Score management

In any game score are maintained to track the efficiency of the player playing the game. This also assist player to maintain a track of the gaming skills and to compare it with other player. This adds excitement and fun to the game.

Score in our game is recorded for two teams during the game play. Team scores when their player destroys walls, bricks, other opponent player or seal. This score is updated on the score board at the end of each level. At the end of the game, out of two scores top score is selected. This top score is then compared with the top 10 scores recorded in the “Scores.dat” file. If it falls within the top 10 scores then users are requested to enter their team name. This information is then saved to the same file for future reference.

Whole purpose of score management is accomplished using following function:

|  |
| --- |
| **Function Name:** Read Score  **Function prototype:** void read\_score();  **Number of Arguments:** 0  **Return Type:** void  **Return Value:** Nothing  **Arguments Description:**  (None)  **Function’s Description:**  This function reads top 10 scores from the file assigning them to the SCORES structure variables of the game.  **Algorithm:**  Algorithm for the read\_score() function is given below:   1. Open the file “Scores.dat” 2. Iterate a loop 10 times and read the scores from the file. 3. Assign the scores to variables in the game created to hold the scores 4. Close the file   **Actual Implementation:**  void read\_score()  {  //Our file pointer  FILE \*fptr;    //For looping  int i=0;    //Open the file Scores.dat  fptr=fopen("Scores","r");    //loop  for(i=0;i<10;i++)  {  //read the file and assign the score to the structure  fread(&score[i],sizeof(SCORES),1,fptr);    }  //close the file  fclose(fptr);  } |
| **Function Name:** Write Score  **Function prototype:** void write\_score(SCORES array[]);  **Number of Arguments:** 1  **Return Type:** void  **Return Value:** Nothing  **Arguments Description:**  Argument 1: A array of 10 SCORES structure variables containing all the top 10 scores.  **Function’s Description:**  This function writes the top 10 scores for the game in the file “Scores.dat”. Top 10 scores are stored in 10 SCORES structure variables in descending order.  **Algorithm:**  Algorithm for the read\_score() function is given below:   1. Open the file “Scores.dat” 2. Iterate a loop 10 times. 3. Write the score information to the file. 4. Close the file   **Actual Implementation:**  void write\_score(SCORES st[])  {  //Our file pointer  FILE \*fptr;    //for looping  int i=0;    //open the file  fptr=fopen("Scores","w");    //loop 10 times  while(i<=10)  {  //write the score information to the file  fwrite(&st[i],sizeof(SCORES),1,fptr);  //increment i  ++i;  }    //close the file  fclose(fptr);  } |
| **Function Name:** Sort Scores  **Function prototype:** void sort\_score();  **Number of Arguments:** 0  **Return Type:** void  **Return Value:** Nothing  **Arguments Description:**  (None)  **Function’s Description:**  This function sorts all the scores stored in the SCORES structure variable using a modified bubble sort. Finally after sorting the variables these variables are writing to the file “Scores.dat”. Sorting is done in the descending order.  **Algorithm:**  Algorithm for the read\_score() function is given below:   1. Iterate a loop 10 times 2. Sort the scores in the loop using bubble sort algorithm 3. Open the file “Scores.dat” 4. Iterate a loop 10 times. 5. Write the sorted score information to the file. 6. Close the file   **Actual Implementation:**  void sort\_score()  {  //Our file pointer  FILE \*fp;  //For looping  int i=0,j;  //Loop for 10 times using bubble sort algorithm  for( i=0;i<10;++i)  {  for(j=i;j<10;++j)  {  if(score[i].score < score[j].score)  {  //swapping  SCORES temp;  temp=score[i];  score[i]=score[j];  score[j]=temp;  }  }  }  //open the file to write sorted scores  fp=fopen("Scores","w");  //loop for writing score to the file  for(i=0;i<10;++i)  {  fwrite(&score[i],sizeof(SCORES),1,fp);  }  //close the file  fclose(fp);  } |

* 1. **User Interface Design**

Interactive gaming involves lots of interaction with the using. This prevents game from being monotonous. Interaction involves taking user response, allowing user to modify setting, enter their name and view the scores etc. User interface is one of the most crucial part of the game design.

We have vested careful research on the design of the user interface and implemented multiple screens with multiple options in order to allow user interaction with the game. Even though there is some limited user interaction with the game. User can still get their name on the high score board, turn on and off sound, look for credits and most of all play the game.

* + 1. Main Welcome Screen

Main welcome screen welcomes the user to the game. It measures 1300 pixels X 600 pixels. It has buttons to provide user with further interaction option. They are described in the table below with their functionality.

|  |  |  |
| --- | --- | --- |
| S.N. | Button Name | Function |
| 1. | Play Game | Start the main game |
| 2. | Options | Enables user to turn sound on and off |
| 3. | Hall of Fame | Shows top 10 scores made in the game |
| 4. | Credits | Displays the names of the developers |
| 5. | Exit | Exits the game |

Table : Button in the main screen with their use

Here is a snapshot of the main game screen:



Figure 7: Snapshot of the main game screen

* + 1. Main Game Screen

This is screen where the actual game begins. All the objects and elements of the game are provided in the screen. The location and placements of wall and bricks are done according to the current level as described in the implementation section. This screen follows the main welcome screen after the user click “Play Game” button. It measures 1100 pixels X 600 pixels.

Here is a snapshot of the main game screen:

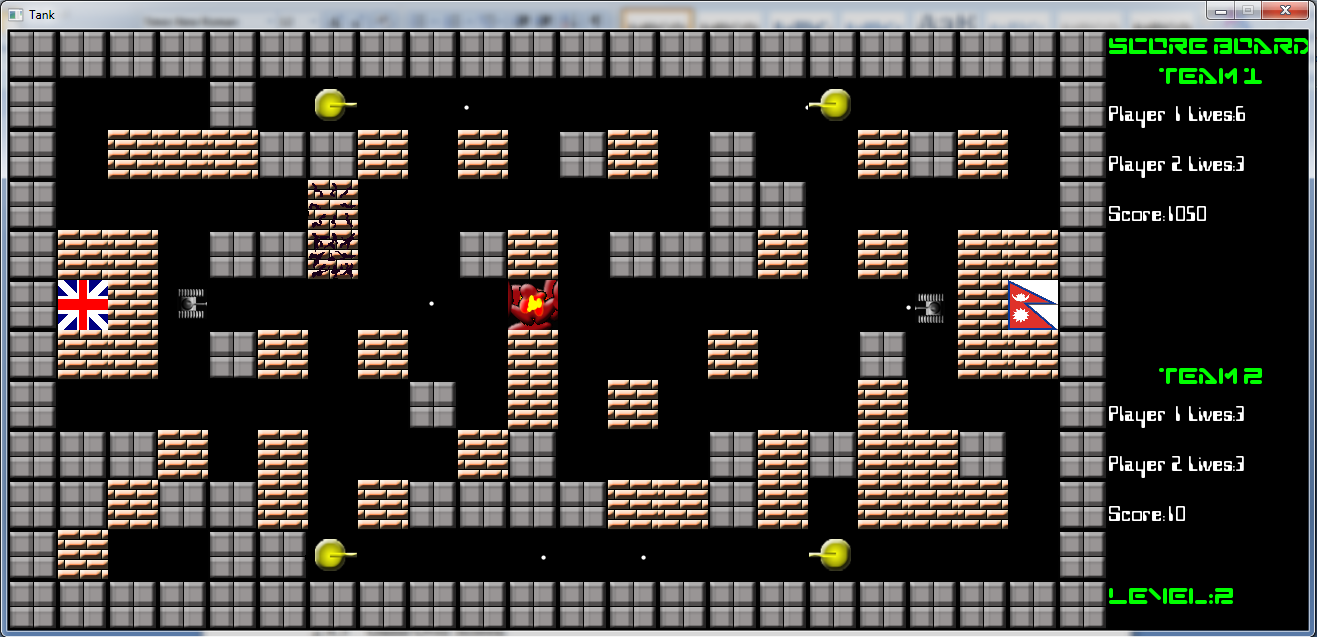


Figure 8: Main game screen with all the game objects

* + 1. Score Board

The Score board maintains the score and player lives information for each team. It is located at the right hand side of the screen and measures 200 pixels X 600 pixels. It can be view in the right side of figure 9. At the bottom it has indication for the current level. This score board get updated each time new level loads.

* + 1. Level Up Screen

This is a small splash screen that shows up every time a new level load. It displays the new level being loaded and a special message to increase user excitement. It holds for 3 seconds before the new level starts.



Figure 9 :Level Up Screen

* + 1. Game Over Screen

This screen is display at the end of the game. It has a dimension of 1300 pixels X 600 pixels.

If the either of the team manages to get the high score it request the user to enter their name through “request user name screen”. It display the congratulation message is team score a high score and give a sympathy message if they fail to do so. Finally it provides user with option to either quit the game or restart the game.

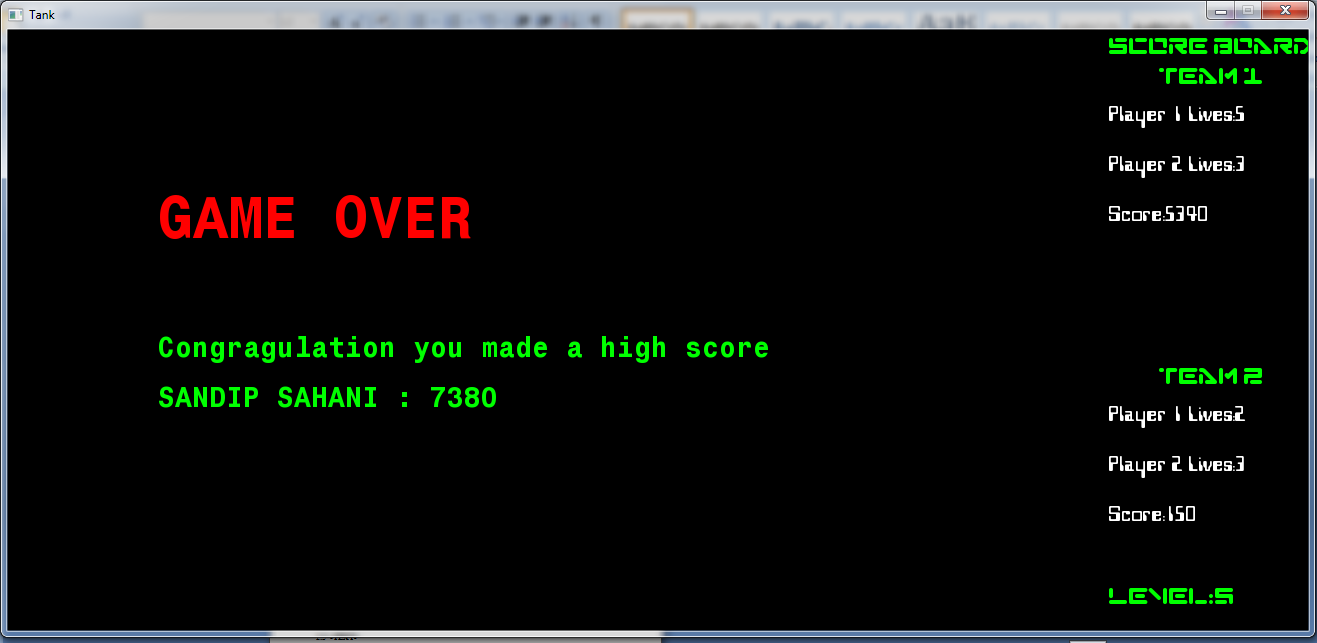


Figure 10 :Game Over Screen displaying high score.

* + 1. Player Name Request Screen

This screen is specially designed to allow user to team their name if they get a high score. This was necessary because even though SDL provide overwhelming amount of features it lack effective user input handling. Thus it was require to custom design such screen.

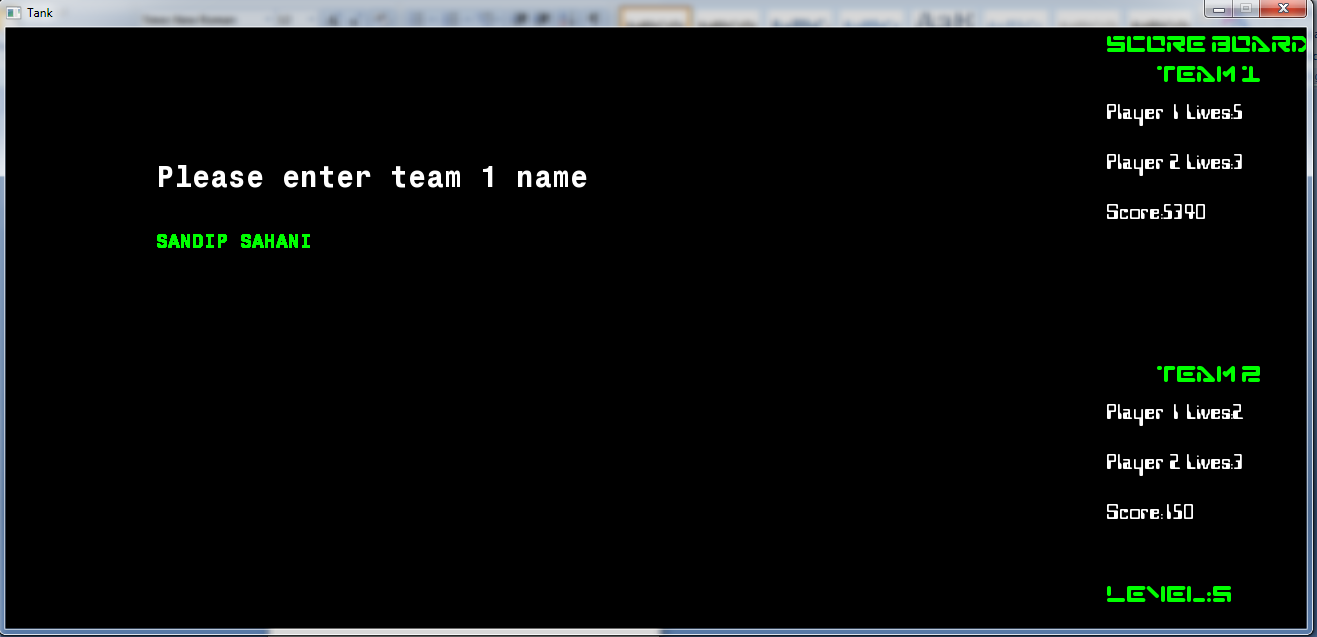


Figure 11: Request name screen asking for team to enter their name

**GANTT CHART**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Work** | **Week 1** | **Week 2** | **Week 3** | **Week 4** | **Week 5** | **Week 6** | **Week 7** | **Week 8** | **Week 9** | **Week 10** |
| Game Analysis and Design | 21 days | | |  | | | | | | |
| Graphics and sound programming |  | 49 days | | | | | | | |  |
| File handling |  | | | | 14 days | |  | | | |
| Event handling and player control |  | 21 days | | |  | | | | | |
| Packaging and testing |  | | | | | | | | | 7 days |

Table 6: Gantt char for the project

Week 1 started from – March 14, 2010

**CONCLUSION**

Our project to develop “The Tank Game” took us through a tour of all possible situations one might encounter in their future undertakings. It was a roller coaster ride all the way. We learned a great deal about programming and team work via this project. We are very grateful to all those beautiful minds and hands that intentionally or unintentionally put a brick in this project, especially to our Supervisor Mr. Pankaj Raj Dawadi for letting us to carry out this project.

This project led us to some serious application development skills that we were totally unfamiliar with. We learnt that undertaking a task of building big software requires a great deal of planning and organization. Unorganized approach leads to disaster. From a prospect of first year computer engineering student we believe that we have given our best effort to come up with something worth it.

We eagerly look forward for more such future endeavors challenging the limitations of our imaginations.

*“Imagination is more important than knowledge”* – Albert Einstein

**BIBLOGRAPHY**

1. Balaguruswamy E “ Programming with C”, Tata McGraw\_Hill,New Delhi
2. Venugopal K.R. and Parsad S.R., “Mastering C”, Tata McGraw-Hill, New Delhi
3. Gehani, N. “An Advanced Introduction ANSI C EDITION” AT&T Bell Labrotories: Murray Hills
4. Prinz, P and Crawford T “ C in a Nutshell” Sherrof Publishers
5. SDL Library resource, [www.libsorg.com/SDL](http://www.libsorg.com/SDL)
6. SDL Tutorials [www.stdltutorial.com/index.php](http://www.stdltutorial.com/index.php)
7. Battle Tank, Online gaming site, [www.y8.com/battletank](http://www.y8.com/battletank)
8. Dev Paks Org, Resources to download Packages SDL Library for Dev Cpp, [www.devpaks.org/SDL](http://www.devpaks.org/SDL)
9. Lazy Foo, Begginers tutorials on SDL Library, [www.lazyfoo.net/SDL/tutorials](http://www.lazyfoo.net/SDL/tutorials)
10. Adhakari et. al, “Payroll System”, Final Year Computer Project, Report No: BE-CE 78, Kathmandu University

**APPENDICES**

**APPENDIX A:**

Our game includes numeric representation and a pictorial representation of each game object. Here are the Images (not to scale) and numeric value for each of the game object.

The numerical value is used to represent object in the level information file.

The Pictorial representation of the object is used in the game.

|  |  |  |
| --- | --- | --- |
| Object Name | Pictorial Representation | Numerical Representation |
| Player | player1_up.bmp | NONE |
| Bullet | bullet.bmp | NONE |
| Enemy | enemy_up.bmp | 9 |
| Wall | wall.bmp | 1 |
| Bricks | brick.bmp | 2 |
| Seal | seal1.bmpseal2.bmp | 5 – player 1  6 – player 2 |
| Empty | NONE | 0 |

Table 7: Table of Game objects with symbolic representation