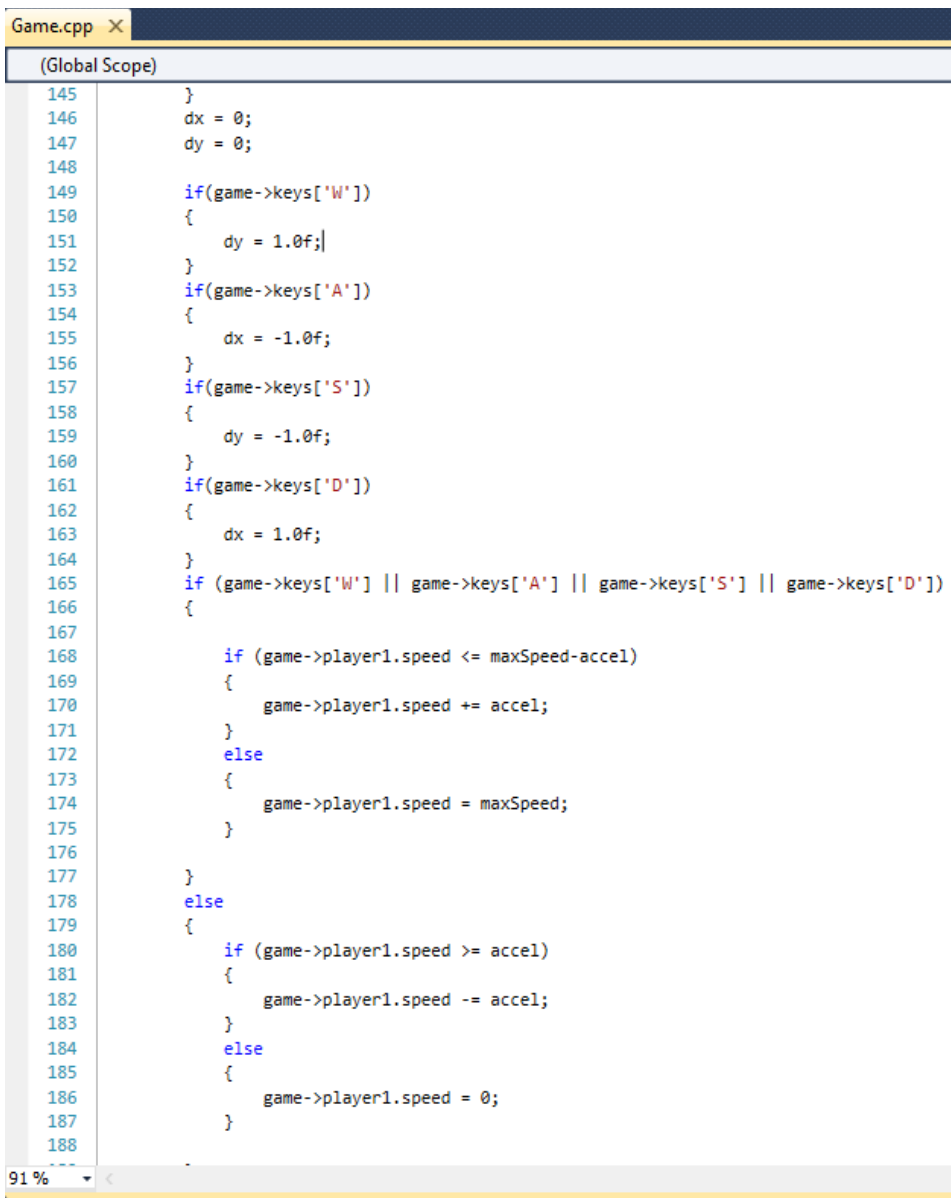


Feature Documentation

The Player

The player is loaded in at spawn point based on the loaded level and from their the user can move using wasd and shoot using the arrow keys. The player can also collect keys and use these to open door.

A screenshot of a code editor window titled 'Game.cpp'. The editor shows C++ code for player movement. The code is line-numbered from 145 to 188. It includes logic for handling 'W', 'A', 'S', and 'D' key presses to update dx and dy values. It also features a speed management system with 'accel' and 'maxSpeed' variables, using if-else statements to increase or decrease speed based on acceleration. The code is written in a dark-themed editor with a light blue header bar.

```
145     }  
146     dx = 0;  
147     dy = 0;  
148  
149     if(game->keys['W'])  
150     {  
151         dy = 1.0f;  
152     }  
153     if(game->keys['A'])  
154     {  
155         dx = -1.0f;  
156     }  
157     if(game->keys['S'])  
158     {  
159         dy = -1.0f;  
160     }  
161     if(game->keys['D'])  
162     {  
163         dx = 1.0f;  
164     }  
165     if (game->keys['W'] || game->keys['A'] || game->keys['S'] || game->keys['D'])  
166     {  
167  
168         if (game->player1.speed <= maxSpeed-accel)  
169         {  
170             game->player1.speed += accel;  
171         }  
172         else  
173         {  
174             game->player1.speed = maxSpeed;  
175         }  
176     }  
177     else  
178     {  
179  
180         if (game->player1.speed >= accel)  
181         {  
182             game->player1.speed -= accel;  
183         }  
184         else  
185         {  
186             game->player1.speed = 0;  
187         }  
188     }
```

This code fields wasd and changes the velocity the player will move when player.move is run

```
Game.cpp X
(Global Scope)
193
194     updateGame(game, dx, dy, secsPassed);
195     if (game->keys[VK_UP])
196     {
197         if(game->player1.curCoolDown ==0 )
198         {
199             addShot(game, game->player1.x, game->player1.y, game->shotTex, entity::UP);
200             game->player1.curCoolDown = game->player1.shotCoolDown;
201         }
202         game->player1.facing = entity::DOWN;
203     }
204     else if (game->keys[VK_LEFT])
205     {
206         if(game->player1.curCoolDown ==0 )
207         {
208             addShot(game, game->player1.x, game->player1.y, game->shotTex, entity::LEFT);
209             game->player1.curCoolDown = game->player1.shotCoolDown;
210         }
211         game->player1.facing = entity::LEFT;
212     }
213     else if (game->keys[VK_RIGHT])
214     {
215         if(game->player1.curCoolDown ==0 )
216         {
217             addShot(game, game->player1.x, game->player1.y, game->shotTex, entity::RIGHT);
218             game->player1.curCoolDown = game->player1.shotCoolDown;
219         }
220         game->player1.facing = entity::RIGHT;
221     }
222     else if (game->keys[VK_DOWN])
223     {
224         if(game->player1.curCoolDown ==0 )
225         {
226             addShot(game, game->player1.x, game->player1.y, game->shotTex, entity::DOWN);
227             game->player1.curCoolDown = game->player1.shotCoolDown;
228         }
229         game->player1.facing = entity::UP;
230     }
231 }
```

Fields the arrow keys and spawns a shot entity at the player facing in the direction of the arrow key. Shots move based on their initial direction and cannot change. There is a cooldown for shots so users have limited fire power

```
Game.cpp X
(Global Scope)
465
466 void updateGame(Game* g , double dx, double dy, long double time)
467 {
468     time *= 20;
469     if(game->player1.speed != 0)
470     {
471         game->player1.move(game, dx, dy, time);
472     }
473
474     enemy* enemytest = g->enemies;
475     for(int i = 0; i < g->enemyArraySize; i++)
476     {
477         if(enemytest->isValid)
478         {
479             enemytest->move(g, g->player1.x - enemytest->x, g->player1.y - enemytest->y, time);
480         }
481         enemytest++;
482     }
483
484     shot* shotTemp = g->shots;
485     for(int i = 0; i < g->shotArraySize; i++)
486     {
487         if(shotTemp->isValid)
488         {
489             shotTemp->move(g, time);
490         }
491         shotTemp++;
492     }
493
494
495
496     if(g->player1.curCoolDown < 0)
497     {
498         g->player1.curCoolDown = 0;
499     }
500     if(g->player1.curCoolDown > 0)
501     {
502         g->player1.curCoolDown -= time;
503     }
504
505
506 }
```

In update game we move the player, enemies and shots as well as cool down the players shot timer. In the game removed enemies are zeroed in the game array so the boolean isValid will be false. This means only entities that have been initialised are rendered and collided.

Animation

Both enemies and the player are animated using 8 images(1 for each animation) that cycle when they move. The current frame is stored in their parent entity.

Game.h	Game.cpp
<pre>8 9 class entity 10 { 11 public: 12 int facing; 13 const static int UP = 0; 14 const static int LEFT = 3; 15 const static int DOWN = 2; 16 const static int RIGHT = 1; 17 18 double x; 19 double y; 20 double height; 21 double width; 22 int texID; 23 double curAnimFrame;</pre>	<pre>928 929 if(xMoveGood yMoveGood) 930 { 931 curAnimFrame += speed*time/4; 932 } 933 if(curAnimFrame > 8) 934 { 935 curAnimFrame = (int)curAnimFrame % 8; 936 }</pre>

Right:
Declaration of current Animation frame

Above:
Update of current Animation frame. Called
in player.move(shown above) and
enemy.move

```

Game.h  Game.cpp X
(Global Scope)
659
660
661 //draw enemies(push and pop per enemy)
662 for(int i = 0; i <= g->enemyArraySize; i++)
663 {
664     glPushMatrix();
665     drawing = &g->enemies[i];
666     if(&g->enemies[i] != 0)
667     {
668         glTranslatef(drawing->x, drawing->y, 0);
669         glRotatef(90 * drawing->facing, 0, 0, 1);
670         glBindTexture(GL_TEXTURE_2D, g->enemyTex[(int)drawing->curAnimFrame]);
671         fillRectangleFromCentre(drawing->width, drawing->height);
672     }
673     glPopMatrix();
674 }

```

when drawing enemies (and player) the texture that is used is gotten from an array of texture handles using the truncated value of the current animation counter.

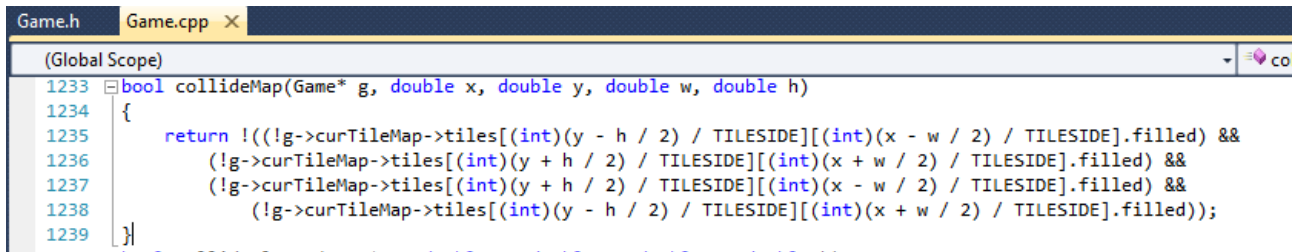
<pre> Game.h Game.cpp X (Global Scope) 319 320 void initGame(Game* g) 321 { 322 g->floorTex = loadPNG("metal_plates.png"); 323 g->wallTex = loadPNG("wall.png"); 324 g->shotTex = loadPNG("shot.png"); 325 g->playerTex[0] = loadPNG("player1.png"); 326 g->playerTex[1] = loadPNG("player2.png"); 327 g->playerTex[2] = loadPNG("player3.png"); 328 g->playerTex[3] = loadPNG("player4.png"); 329 g->playerTex[4] = loadPNG("player5.png"); 330 g->playerTex[5] = loadPNG("player6.png"); 331 g->playerTex[6] = loadPNG("player7.png"); 332 g->playerTex[7] = loadPNG("player8.png"); 333 g->enemyTex[0] = loadPNG("enemy1.png"); 334 g->enemyTex[1] = loadPNG("enemy2.png"); 335 g->enemyTex[2] = loadPNG("enemy3.png"); 336 g->enemyTex[3] = loadPNG("enemy4.png"); 337 g->enemyTex[4] = loadPNG("enemy5.png"); 338 g->enemyTex[5] = loadPNG("enemy6.png"); 339 g->enemyTex[6] = loadPNG("enemy7.png"); 340 g->enemyTex[7] = loadPNG("enemy8.png"); 341 g->doorTex = loadPNG("door.png"); 342 g->stairTex = loadPNG("stairs.png"); 343 g->keyTex = loadPNG("key.png"); 344 </pre>	<p>Textures are loaded in the init game function which is called before the start of the game loop. Static textures are stored as int and textures for animated entities are stored in an int array.</p>
---	--

Collision

Collision is done using a collection of collide functions for each different type of object in the game(map, player, shots, enemies, doors and stairs).

```
Game.h X Game.cpp
entity
156 door* collideDoors(Game* g, double x, double y, double w, double h);
157 bool collideStairs(Game* g, double x, double y, double w, double h);
158 enemy* collideEnemies(Game* g, double x, double y, double w, double h);
159 bool collideShots(Game* g, double x, double y, double w, double h);
160 bool collideMap(Game* g, double x, double y, double w, double h);
161 bool collidePlayer(Game* g, double x, double y, double w, double h);
, double w, double h);
Game.h X Game.cpp
entity
90
91 struct Game
92 {
93     int floorTex;
94     int wallTex;
95     int shotTex;
96     int playerTex[8];
97     int enemyTex[8];
98     int doorTex;
99     int stairTex;
100     int keyTex;
101     bool done;
102     bool keys[256];
103     double mouseXPos;
104     double mouseYPos;
105
106     Tilemap tileMaps[16];
107     Tilemap* curTileMap;
108     int levelCount;
109     int curLevel;
110
111     stair stairs;
112
113     door doors[200];
114     const static int doorArraySize = 200;
115
116     player player1;
117
118     enemy enemies[200];
119     const static int enemyArraySize = 200;
120
121     key gameKeys[200];
122     const static int keyArraySize = 200;
123
124     shot shots[200];
125     const static int shotArraySize = 200;
126
127
128
129 };
```

Collision with the map is done by integer dividing the passed x, y and x+w and y + h by the tileSize(16) and using the value to access the tile this x and y sit on. If the tile is denoted as filled the function returns true;



The screenshot shows a code editor with two tabs: 'Game.h' and 'Game.cpp'. The 'Game.cpp' tab is active, showing the implementation of the `collideMap` function. The function signature is `bool collideMap(Game* g, double x, double y, double w, double h)`. The function body consists of a single return statement that checks if the player's bounding box overlaps with any filled tiles on the map. The logic is as follows: it returns `true` if any of the four corners of the player's bounding box (defined by `x, y, x+w, y+h`) overlaps with a filled tile. The tiles are accessed using integer division by `TILESIDE` (16). The function returns `false` if no overlap is found.

```
1233 bool collideMap(Game* g, double x, double y, double w, double h)
1234 {
1235     return !((!g->curTileMap->tiles[(int)(y - h / 2) / TILESIDE][(int)(x - w / 2) / TILESIDE].filled) &&
1236             (!g->curTileMap->tiles[(int)(y + h / 2) / TILESIDE][(int)(x + w / 2) / TILESIDE].filled) &&
1237             (!g->curTileMap->tiles[(int)(y + h / 2) / TILESIDE][(int)(x - w / 2) / TILESIDE].filled) &&
1238             (!g->curTileMap->tiles[(int)(y - h / 2) / TILESIDE][(int)(x + w / 2) / TILESIDE].filled));
1239 }
```

Collision with everything else is done using axis aligned bound boxes. Each collision method is largely the same but for instance if its enemies or keys it loops through all them and checks each but for player or stairs it doesn't need to loop as only one exists.

```

Game.h  Game.cpp X
(Global Scope)
1240 bool collidePlayer(Game* g, double x, double y, double w, double h)
1241 {
1242     player* test = &g->player1;
1243
1244     if(test->isValid)
1245     {
1246         if (!(y > test->y + test->height ||
1247             y + h < test->y ||
1248             x > test->x + test->width ||
1249             x + w < test->x))
1250         {
1251             return true;
1252         }
1253     }
1254
1255     return 0 ;
1256 }
1257
1258 key* collideKeys(Game* g, double x, double y, double w, double h)
1259 {
1260     key* test = g->gameKeys;
1261     for (int i = 0; i < g->keyArraySize; i++)
1262     {
1263         if(!(x == test->x && y == test->y))
1264         {
1265             if(test->isValid)
1266             {
1267                 if (!(y > test->y + test->height ||
1268                     y + h < test->y ||
1269                     x > test->x + test->width ||
1270                     x + w < test->x))
1271                 {
1272                     return test;
1273                 }
1274             }
1275             test++;
1276         }
1277     }
1278     return 0;
1279 }

```

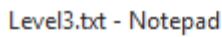
Acceleration

The players movement in the game is not instantaneous and despite it not taking long his speed does increase at the start of his movement.

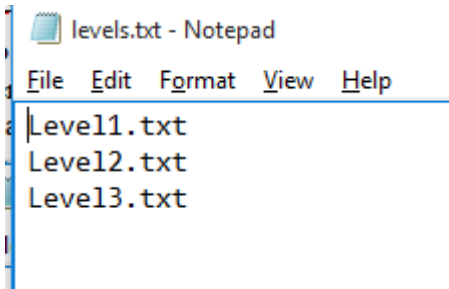

```
Game.h  Game.cpp X
(Global Scope)
165     if (game->keys['W'] || game->keys['A'] || game->keys['S'] || game->keys['D'])
166     {
167
168         if (game->player1.speed <= maxSpeed-accel)
169         {
170             game->player1.speed += accel;
171         }
172         else
173         {
174             game->player1.speed = maxSpeed;
175         }
176
177     }
178     else
179     {
180         if (game->player1.speed >= accel)
181         {
182             game->player1.speed -= accel;
183         }
184         else
185         {
186             game->player1.speed = 0;
187         }
188
189     }
190
191
192
Game.h  Game.cpp X
player
830 void player::move(Game* g, double dx, double dy, double time)
831 {
832     bool yMoveGood = false;
833     bool xMoveGood = false;
834     double newX;
835     double newY;
836     double length = sqrt(dx*dx + dy*dy);
837     if(length){
838         newX = x + (dx*speed*time/length);
839         newY = y + (dy*speed*time/length);
840     }
841     else
842     {
843         newX = x;
844         newY = y;
845     }
846
847
```

LevelSets and TileMaps

To make it easy to store levels in the filesystem each level is stored as a csv file using integers to denote what should be done for that tile(0 = floor, 1 = wall, 2 = floor + spawn enemy, 3 = floor + spawn player, 4 = floor + spawn door and 5 = floor + spawn stairs).

[illegible]

Kyle Alexander 100082709



Level Editor

The level editor is copied from the source of the game with a few changes. The ai and movement code as been commented out or deleted and mouse controls added. Clicking on a tile increments its tile id mod 6 and then reloads the tilemap. When the user launches the editor they are presented with a tile map which is four walls and floor with nothing in it. Once the user has set up their tilemap they hit space and it writes the tile ids to a file called newLevel.txt which they can rename and add to levels.txt as explained before.

This game.cpp is editor/game.cpp where all above are Graphicsgame/game.cpp. Ill note whenever it is editor/game.cpp

```
Game.cpp x
(Global Scope)
46
47     case WM_LBUTTONDOWN:
48     {
49         SetCursor(NULL);
50         cycleTile(game, game->mouseXPos, game->mouseYPos);
51     }
52     break;
53
54     case WM_LBUTTONUP:
55     {
56     }
57     break;
58
59     case WM_MOUSEMOVE:
60     {
61
62         game->mouseXPos = (LOWORD(lParam) - windowWidth/2)/camZoom + camX;
63         game->mouseYPos = ((windowHeight - HIWORD(lParam)) - windowHeight/2)/camZoom + camY;
64     }
```

Mouse Controls


```

void cycleTile(Game* g, float x, float y)
{
    Tile* t;
    if(!(x < 0 || y < 0))
    {
        t = &g->curTileMap->tiles[(int)y/TILESIDE][(int)x/TILESIDE];
        t->tileID++;
        if(t->tileID == 6)
            t->tileID = 0;
        refreshTileMap(g);
    }
}

```

change tile id based on the ingame location of the users click

Time Stepping

```
Game.cpp X
(Global Scope)
127
128     QueryPerformanceFrequency(&cps);
129     double secsPassed = 0;
130     QueryPerformanceCounter(&curCount);
131     while(!game->done)
132     {
133         prevCount = curCount;
134         QueryPerformanceCounter(&curCount);
135         countDifference = curCount.QuadPart - prevCount.QuadPart;
136         secsPassed = (long double)countDifference / (long double)cps.QuadPart;
```

The seconds passed between frames is calculated at the start of each loop and used in the move functions of all the entities

```
Game.h    Game.cpp X
player
830 void player::move(Game* g, double dx, double dy, double time)
831 {
832     bool yMoveGood = false;
833     bool xMoveGood = false;
834     double newX;
835     double newY;
836     double length = sqrt(dx*dx + dy*dy);
837     if(length){
838         newX = x + (dx*speed*time*length);
839         newY = y + (dy*speed*time*length);
840     }
841     else
842     {
843         newX = x;
844         newY = y;
845     }
846
```

Scoring

If a player finishes the game their score is the total number of keys they collected. This score ignores keys spent on doors. This score is presented upon reaching the end of the final level.

```
Game.h  Game.cpp X
player
896
897     if(key* k = collideKeys(g, x, y, width, height))
898     {
899         keyCount++;
900         score++;
901         removeKey(g, k);
902     }
```

Add 1 to your score

```
Game.h  Game.cpp X
player
937         curAnimFrame = (int)curAnimFrame % 6;
938     }
939     if(collideStairs(g, x, y, width, height))
940     {
941
942
943         if(g->curLevel == g->levelCount)
944         {
945             char* winmsg = "";
946             sprintf(winmsg, "you win. you collected %d keys. well done", score);
947             MessageBox(NULL, winmsg, "WINNNNNNNERNERNERNERN", MB_OK);
948             g->done = true;
949         }
950         else
951         {
952             setWorld(g, g->curLevel + 1);
953         }
954     }
955 }
```

Display score when game finishes