

# Pacenstein

- Technisch Verslag -

02/02/2022



Lennard Duinkerken  
Emma Raijmakers  
Daan Roth  
Jarno Bröcker

# Contents

<b>1</b>	<b>Inleiding</b>	<b>2</b>
1.1	Klassendiagram . . . . .	2
<b>2</b>	<b>Game Engine</b>	<b>3</b>
2.1	Raycasting . . . . .	3
2.1.1	Waarom Raycasting . . . . .	3
2.1.2	Hoe werkt Raycasting - In het kort . . . . .	3
2.1.3	Hoe werkt Raycasting - In het lang . . . . .	4
2.1.4	Port en Origineel . . . . .	4
2.1.5	Grote Verschillen . . . . .	4
2.2	Game Logic . . . . .	7
2.2.1	Player Movement . . . . .	7
2.2.2	Player Interaction . . . . .	8
2.2.3	Ghost Movement . . . . .	8
2.2.4	Optimalisatie . . . . .	9
2.3	Asset Manager . . . . .	10
2.4	Input Manager . . . . .	10
2.5	State Machine . . . . .	10
<b>3</b>	<b>States</b>	<b>11</b>
3.1	Splash . . . . .	11
3.2	Menu's . . . . .	11
3.2.1	Main Menu . . . . .	11
3.2.2	Settings Menu . . . . .	12
3.2.3	Leaderboard Menu . . . . .	13
3.2.4	Credits Menu . . . . .	13
3.3	In Game . . . . .	14
3.4	Hunting . . . . .	14
3.5	Pause State . . . . .	14
3.6	Game Over . . . . .	15
<b>4</b>	<b>Entities</b>	<b>17</b>
4.1	Items . . . . .	17
4.1.1	Fruits . . . . .	17
4.1.2	Ghosts . . . . .	17
4.2	Player . . . . .	17
<b>5</b>	<b>Resources</b>	<b>18</b>
5.1	UI Elementen . . . . .	18
5.2	Textures . . . . .	18
5.3	Sprites . . . . .	18
5.4	Afbeeldingen . . . . .	18
5.5	Overige . . . . .	18
<b>6</b>	<b>Toekomstplannen</b>	<b>19</b>
6.1	Ghosts Bewegen . . . . .	19
6.2	Fruits . . . . .	19
6.3	Map Editor . . . . .	19
6.4	De Moeilijkheidsgraad . . . . .	19
6.5	Keybindings . . . . .	19
6.6	Visuele Feedback . . . . .	20



## 2. Game Engine

### 2.1 Raycasting

#### 2.1.1 Waarom Raycasting

Dit project gebruikt raycasting om een 2d wereld, op het scherm er uit te laten zien als een 3d wereld. Raycasting is hiervoor gebruikt omdat het een relatief simpele methode is om de wereld 3d te maken. Dit project gebruikt geen hardware acceleratie, dus het is belangrijk dat het een lichte simpele methode is. Vandaar de keuze voor raycasting.

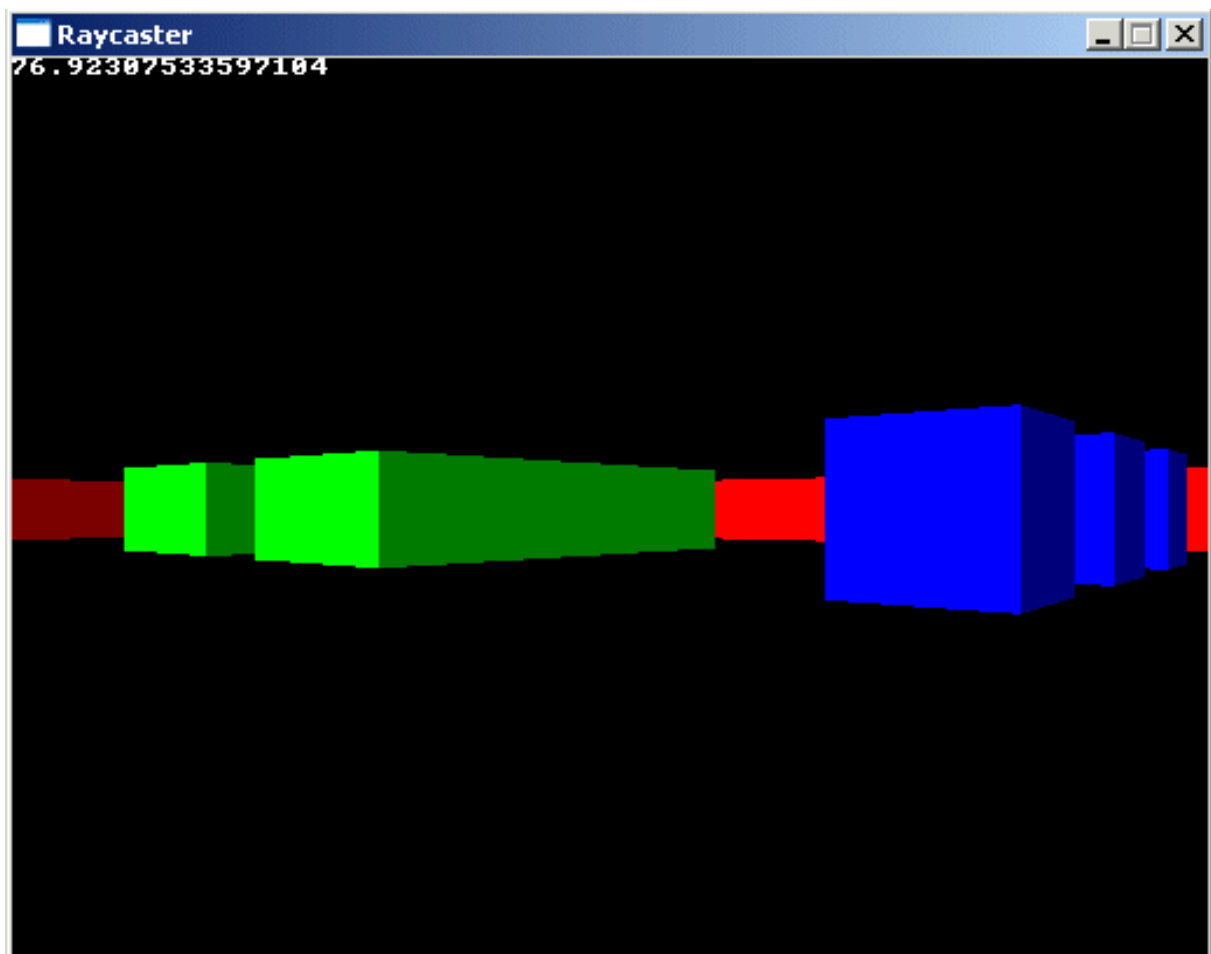


Figure 2.1: Source: <https://lodev.org/cgtutor/raycasting.html>

#### 2.1.2 Hoe werkt Raycasing - In het kort

Raycasting is een erg snel process, en kan op heel low end hardware draaien. Denk aan de originele wolfenstein of doom. De player schiet voor elke pixel van het scherm in de breedte, een ray uit. Hij cast dus een ray. Deze ray moet een eindje reizen vaak. Wanneer deze een muur raakt, zal hij kijken hoe ver hij heeft moeten reizen. Hij gebruikt deze afstand om op het scherm een lijn te trekken. Deze lijn kan groter en kleiner worden gemaakt gebaseerd op de afstand. Wanneer dit process herhaald wordt voor elke pixel, zal er een muur zichtbaar moeten zijn.

Dit is de raycaster in een notendop, en een erge versimpeling. Bij de goed werkende raycaster komt veel meer kijken. Heb je de muur van voren geraakt? Of heb je de muur aan de

zijkant geraakt? Kan je dit gebruiken om een nep schaduw op een van de kanten te zetten zodat je het verschil ziet tussen de muren?

### 2.1.3 Hoe werkt Raycasting - In het lang

Hiervoor is <https://lodev.org/cgtutor/raycasting.html> een goede bron met uitleg. Deze website loopt stap voor stap af hoe een raycaster gemaakt moet worden. En enkele optimalisaties. De raycaster van de game is gebaseerd op hoofdstuk 1 en 3.

### 2.1.4 Port en Origineel

De raycaster van de engine is een directe port van <https://lodev.org/cgtutor/raycasting.html>. De comments van de originele broncode zijn er bij gehouden, en variabelen hebben dezelfde naam waar dit mogelijk is.

Het grootste verschil tussen de port en het origineel is het feit dat het origineel SDL gebruikt, en de port SFML. Ook is de code wat aangepast zodat deze meer volgens de C++ standaarden werkt. Bijvoorbeeld de c-arrays waar de map is in opgeslagen zijn veranderd in `std::array`.

De raycaster is opgedeelt in 2 losse raycasters. 1 voor de muren, en 1 voor de entities. De raycaster voor de entities heeft de meeste veranderingen gekregen vergeleken met het origineel in verband met SFML. Deze 2 raycasters worden in de `InGameState.cpp` en `InGameState.hpp` gemaakt en uitgevoerd. Dit gebeurt in de draw functie.

### 2.1.5 Grote Verschillen

#### Movement

##### Origineel

```
1 readKeys();
2 if (keyDown(SDLK_UP)) {
3     if(worldMap[int(posX + dirX * moveSpeed)][int(posY)] == false) posX += dirX * moveSpeed;
4     if(worldMap[int(posX)][int(posY + dirY * moveSpeed)] == false) posY += dirY * moveSpeed;
5 }
6 if (keyDown(SDLK_DOWN)) {
7     if(worldMap[int(posX - dirX * moveSpeed)][int(posY)] == false) posX -= dirX * moveSpeed;
8     if(worldMap[int(posX)][int(posY - dirY * moveSpeed)] == false) posY -= dirY * moveSpeed;
9 }
10 //rotate to the right
11 if (keyDown(SDLK_RIGHT)) {
12     double oldDirX = dirX;
13     dirX = dirX * cos(-rotSpeed) - dirY * sin(-rotSpeed);
14     dirY = oldDirX * sin(-rotSpeed) + dirY * cos(-rotSpeed);
15     double oldPlaneX = planeX;
16     planeX = planeX * cos(-rotSpeed) - planeY * sin(-rotSpeed);
17     planeY = oldPlaneX * sin(-rotSpeed) + planeY * cos(-rotSpeed);
18 }
19 //rotate to the left
20 if (keyDown(SDLK_LEFT)) {
21     double oldDirX = dirX;
22     dirX = dirX * cos(rotSpeed) - dirY * sin(rotSpeed);
23     dirY = oldDirX * sin(rotSpeed) + dirY * cos(rotSpeed);
24     double oldPlaneX = planeX;
25     planeX = planeX * cos(rotSpeed) - planeY * sin(rotSpeed);
26     planeY = oldPlaneX * sin(rotSpeed) + planeY * cos(rotSpeed);
27 }
```

## Port

```
1 sf::Event event;
2
3 while (this->data->window.pollEvent(event)) {
4     this->data->window.setMouseCursorVisible(false);
5
6     if (sf::Event::Closed == event.type) this->data->window.close();
7
8     if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_UP)
9     || sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_ALT_UP)
10    || sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_DOWN)
11    || sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_ALT_DOWN)
12    || sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_RIGHT)
13    || sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_ALT_RIGHT)
14    || sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_LEFT)
15    || sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_ALT_LEFT)) {
16        this->data->machine.addState(state_ref_t(std::make_unique<HuntingState>(this->data)), true);
17    }
18
19    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_PAUSE)
20    || sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_ALT_PAUSE)) {
21        this->generatePauseBackground();
22        this->data->machine.addState(state_ref_t(std::make_unique<PauseState>(this->data)), false);
23    }
24
25    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_EXIT)) {
26        this->data->window.close();
27    }
28 }
```

## Muren tekenen

### Origineel

```
1 if(side == 0) perpWallDist = (sideDistX - deltaDistX);
2 else         perpWallDist = (sideDistY - deltaDistY);
```

### Port

```
1 if (!side) {
2     perpWallDist = fabs((mapX - rayPosX + (1 - stepX) / 2) / rayDirX);
3     while (rayPosY > 1) {
4         rayPosY = -1;
5     }
6     textureSectionX = rayPosY;
7 }
8 else {
9     perpWallDist = fabs((mapY - rayPosY + (1 - stepY) / 2) / rayDirY);
10
11 while (rayPosX > 1) rayPosX -= 1;
12 textureSectionX = rayPosX;
13 }
```

## Sprites sorteren

### Origineel

```
1 void sortSprites(int* order, double* dist, int amount) {
2     std::vector<std::pair<double, int>> sprites(amount);
3     for(int i = 0; i < amount; i++) {
4         sprites[i].first = dist[i];
5         sprites[i].second = order[i];
6     }
7     std::sort(sprites.begin(), sprites.end());
8     for(int i = 0; i < amount; i++) {
9         dist[i] = sprites[amount - i - 1].first;
10        order[i] = sprites[amount - i - 1].second;
11    }
12 }
```

### Port

```
1 void InGameState::sortSprites(std::vector<int> &order, std::vector<float> &dist, int size) {
2     std::cout << "Done \n";
3     int gap = size;
4     bool flag = true;
5     while (gap != 1 || flag) {
6         gap = (gap * 10) / 13;
7         if (gap < 1) gap = 1;
8         flag = false;
9
10        for (int i = 0; i < size - gap; i++) {
11            int j = i + gap;
12            if (dist[i] < dist[j]) {
13                std::swap(dist[i], dist[j]);
14                std::swap(order[i], order[j]);
15                flag = true;
16            }
17        }
18    }
19 }
```

## Sprites tekenen

### Origineel

```
1 //loop through every vertical stripe of the sprite on screen
2 for(int stripe = drawStartX; stripe < drawEndX; stripe++) {
3     int texX = int(256 * (stripe - (-spriteWidth / 2 + spriteScreenX)) * texWidth / spriteWidth) / 256;
4
5     //the conditions in the if are:
6     //1) it's in front of camera plane so you don't see things behind you
7     //2) it's on the screen (left)
8     //3) it's on the screen (right)
9     //4) ZBuffer, with perpendicular distance
10
11    if(transformY > 0 && stripe > 0 && stripe < w && transformY < ZBuffer[stripe]) {
12        for(int y = drawStartY; y < drawEndY; y++) { //for every pixel of the current stripe
13
14            //256 and 128 factors to avoid floats
15            int d = (y-vMoveScreen) * 256 - h * 128 + spriteHeight * 128;
16            int texY = ((d * texHeight) / spriteHeight) / 256;
17
18            //get current color from the texture
19            Uint32 color = texture[sprite[spriteOrder[i]].texture][texWidth * texY + texX];
20
21            //paint pixel if it isn't black, black is the invisible color
22            if((color & 0x00FFFFFF) != 0) buffer[y][stripe] = color;
23        }
24    }
25 }
26 }
```

## Port

```
1 //loop through every vertical stripe of the sprite on screen
2 for (int stripe = drawStartX; stripe <= drawEndX; ++stripe) {
3     if (stripe < 0) {
4         drawStartX += 1;
5         continue;
6     }
7     // are other walls in front
8     if (transformY > this->ZBuffer[stripe]) {
9         drawStartX += 1;
10        continue;
11    }
12    break;
13 }
14
15 int newWidth = drawEndX - drawStartX;
16 float d = (float)drawWidth / (float)newWidth;
17
18 spriteLeft = texture_width - (float)texture_width / d;
19
20 for (int stripe = drawEndX; stripe > drawStartX; --stripe) {
21     if (stripe > width) {
22         drawEndX -= 1;
23         continue;
24     }
25     if (transformY > this->ZBuffer[stripe]) {
26         drawEndX -= 1;
27         continue;
28     }
29     break; // y break, de continues snap ik, maar deze is niet nodig
30 }
31
32 newWidth = drawEndX - drawOrigStartX;
33 d = (float)drawWidth / (float)newWidth;
34
35 spriteRight = (float)texture_width / d;
36
37 sf::VertexArray spriteQuad(sf::Quads, 4);
38
39 spriteQuad[0].position = sf::Vector2f(drawStartX, drawStartY);
40 spriteQuad[0].texCoords = sf::Vector2f(spriteLeft, spriteTop);
41 spriteQuad[1].position = sf::Vector2f(drawEndX, drawStartY);
42 spriteQuad[1].texCoords = sf::Vector2f(spriteRight, spriteTop);
43 spriteQuad[2].position = sf::Vector2f(drawEndX, drawEndY);
44 spriteQuad[2].texCoords = sf::Vector2f(spriteRight, spriteBottom);
45 spriteQuad[3].position = sf::Vector2f(drawStartX, drawEndY);
46 spriteQuad[3].texCoords = sf::Vector2f(spriteLeft, spriteBottom);
47
48 this->data->window.draw(spriteQuad, &sprite.tex);
```

## 2.2 Game Logic

### 2.2.1 Player Movement

De Player class is gebaseerd op de Entity, en heeft daarom `sf::Vector2f` s om zijn position en direction op te slaan.

#### Entity.hpp

```
1 protected:
2     sf::Vector2f position, direction, size;
3     float moveSpeed;
4 };
```

Deze wordt door de player input aangepast met zijn input, de handleInput functie. Deze veranderd de state met de juiste waarden.



## InGameState.hpp

```
1 void InGameState::handleInput() {
2     sf::Event event;
3
4     while (this->data->window.pollEvent(event)) {
5         this->data->window.setMouseCursorVisible(false);
6
7         if (sf::Event::Closed == event.type) this->data->window.close();
8
9         if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_UP)
10         || sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_ALT_UP)
11         || sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_DOWN)
12         || sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_ALT_DOWN)
13         || sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_RIGHT)
14         || sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_ALT_RIGHT)
15         || sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_LEFT)
16         || sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_ALT_LEFT)) {
17             this->data->machine.addState(state_ref_t(std::make_unique<HuntingState>(this->data)), true);
18         }
19
20         if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_PAUSE)
21         || sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_ALT_PAUSE)) {
22             this->generatePauseBackground();
23             this->data->machine.addState(state_ref_t(std::make_unique<PauseState>(this->data)), false);
24         }
25
26         if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::KEY_EXIT)) {
27             this->data->window.close();
28         }
29     }
30 }
```

### 2.2.2 Player Interaction

De player kan met andere Items in de game interacteren. Dit wordt gedaan door de volgende functie:

```
1 bool Player::intersect(Item & other, game_data_ref_t data){
2     bool collision = int(this->position.x) == int(other.getPos().x)
3     && int(this->position.y) == int(other.getPos().y);
4     if (collision) other.interact(this->data);
5     return collision;
6 }
```

In deze functie wordt gecheckt of de speler op hetzelfde block van de map staat als het meegegeven item. Als dit het geval is, dan is er een collision. Als er een collision is dan wordt de interact functie van het Item waar de speler mee collide aangeroepen. In deze interact functie wordt verder afgehandeld wat er gebeurt bij de collision met een bepaald Item.

```
1 void PowerPellet::interact(game_data_ref_t data) {
2     data->powerPelletsLeft--;
3     collected = true;
4     data->score += points;
5     data->scattering = true;
6 }
```

Een voorbeeld van een interact functie is de interact functie van de PowerPellet class. Hierin wordt aangegeven dat de pellet is verzameld, de score is verhoogd en scattering (als de speler de ghosts kan opeten) actief is.

De interact functie is een abstracte functie van de Item class, waar alle items van erven. Hierdoor wordt het voor iedere class die erft van een Item, en dus ook als item in het spel voor kan komen, verplicht om aan te geven hoe het interact met de player.

### 2.2.3 Ghost Movement

De movement van de ghosts is zo geïmplementeerd dat het per ghost kan verschillen. De Ghost class heeft een abstracte move functie, zodat iedere ghost zijn eigen movement moet regelen.

Een voorbeeld is de move functie van de Blinky class:

```
1  sf::Vector2f Blinky::move(const map_t & worldMap){
2      switch (this->direction) {
3          case 1: // Noord
4              if((this->position.y - int(this->position.y)) < 0.5){
5                  if(worldMap[int(this->position.x)][int(this->position.y) - 1] == 3) {
6                      //if block above ghost is a wall
7                      this->direction = rand() % 4 + 1;
8                      break;
9                  }
10             }
11             this->position = {this->position.x, this->position.y - movement_speed};
12             break;
13
14             case 2: // Oost
15                 if((this->position.x - int(this->position.x)) < 0.5){
16                     if(worldMap[int(this->position.x) - 1][int(this->position.y)] == 3) {
17                         //if block left of ghost is a wall
18                         this->direction = rand() % 4 + 1;
19                         break;
20                     }
21                 }
22                 this->position = {this->position.x - movement_speed, this->position.y};
23                 break;
24
25                 case 3: // Zuid
26                     if((this->position.y - int(this->position.y)) > 0.5){
27                         if(worldMap[int(this->position.x)][int(this->position.y) + 1] == 3) {
28                             //if block below ghost is a wall
29                             this->direction = rand() % 4 + 1;
30                             break;
31                         }
32                     }
33                     this->position = {this->position.x, this->position.y + movement_speed};
34                     break;
35
36                     case 4: // West
37                         if((this->position.x - int(this->position.x)) > 0.5){
38                             if(worldMap[int(this->position.x) + 1][int(this->position.y)] == 3) {
39                                 //if block right of ghost is a wall
40                                 this->direction = rand() % 4 + 1;
41                                 break;
42                             }
43                         }
44                         this->position = {this->position.x + movement_speed, this->position.y};
45                         break;
46                 }
47             return this->position;
48         }
```

De functie berekent de nieuwe positie van Blinky als een `sf::Vector2f`. Dit wordt gedaan door de richting die Blinky op gaat te gebruiken. Er wordt gekeken of Blinky in dezelfde richting door kan gaan of dat er een muur voor hem staat. Als er een muur staat, wordt de richting van Blinky aangepast naar een random nieuwe richting. De nieuwe positie van Blinky wordt berekend door de movement speed op de juiste manier op te tellen bij de oude positie. Deze nieuwe positie wordt teruggegeven.

## 2.2.4 Optimalisatie

In de game is optimalisatie toegepast door middel van de InRange functie in de Player class. In deze functie wordt gekeken of de positie van een Item binnen een vierkant van grootte 5 om de speler valt. Als dit het geval is, wordt er true teruggegeven en anders false.

```
1  bool Player::InRange(Item & other){
2      sf::RectangleShape playerRect;
3      playerRect.setPosition(sf::Vector2f(this->position.x - 5, this->position.y - 5));
4      playerRect.setSize(sf::Vector2f(10.0, 10.0));
5      return playerRect.getGlobalBounds().contains(other.getPos());
6  }
```

De returnwaarde van deze functie wordt gebruikt bij het tekenen van de Pac en Power Pellets. De pellets worden alleen op de map weergegeven als ze in de range van de speler zitten. Dit zorgt ervoor dat er meer pellets aan de map toegevoegd kunnen worden zonder dat dit de performance vermindert.

## 2.3 Asset Manager

De asset manager is verantwoordelijk voor het beheren van de textures, map, fonts, afbeeldingen, configuratiebestand, leaderboard bestand en SFML vertices.

De map wordt ingeladen van een .png-bestand. Deze wordt door de asset manager omgezet naar een vector van vectoren. Door middel van de verschillende kleuren pixels zullen de vectoren hun waarden krijgen. In deze vectoren staat aangegeven op welke locatie welke muur geschreven moet worden. Deze wordt ingeladen door de `assetManager::loadImage` functie.

De configuratie van de game staat in een settings.conf bestand. Deze wordt ingeladen door de `assetManager::getConfigFile` functie.

Het leaderboard wordt ingeladen door de `assetManager::loadCsvFile` functie, en geeft deze een naam voor gemakkelijk gebruik.

De textures worden ingeladen door de `assetManager::loadTexture` functie.

De fonts worden ingeladen door de `assetManager::loadFonts` functie.

De SFML vertexes worden ingeladen door `assetManager::loadVertex`.

De afbeeldingen worden ingeladen en gegeven door de `assetManager::getImage` function. Deze wordt teruggegeven in een map met de naam en de image zelf.

## 2.4 Input Manager

De input manager houdt de positie van de muiscursor bij en controleert of sprites aangeklikt zijn. Een aantal van onze sprites werken namelijk als knoppen in de menus.

## 2.5 State Machine

De state machine beheert de states door ze op een stack te zetten. De bovenste state is altijd de actieve state. Wanneer een state toegevoegd wordt komt deze bovenop te staan en wordt deze dus ook actief.

Het is mogelijk een state toe te voegen en die state de huidige actieve state laten vervangen. Het vervangen van een state is nuttig wanneer je niet terug wilt gaan naar de huidige state. Als hij niet vervangen wordt, dan kun je de bovenste state verwijderen om terug te keren naar de vorige state.

De state machine is in staat om states te pauzeren en te laten vervolgen en gaat daar mee bezig wanneer `stateMachine::processStateChange` aangeroepen wordt. Het is ook mogelijk om de huidige actieve state op te vragen via `stateMachine::getActiveState`.

## 3. States

De states worden gebruikt om tussen verschillende delen van de game te switchen. Dit wordt gedaan door een state op de stack van de State Machine te zetten of hem er weer af te halen. Iedere state heeft zijn eigen functionaliteiten en graphical interface die in de volgende hoofdstukken verder worden uitgelegd.

### 3.1 Splash

De Splash State wordt gebruikt om alle afbeeldingen die gebruikt worden in de game te laden als Textures. De afbeeldingen worden door de Asset Manager opgeslagen en kunnen in de rest van de game gebruikt worden, zonder dat ze opnieuw geladen moeten worden.



Figure 3.1: Het splash screen - Toont het logo terwijl het assets laadt.

### 3.2 Menu's

De game Pacenstein heeft verschillende menus die informatie aan de speler laten zien. De menus beschikken over tekstelementen en knoppen. Knoppen zijn klikbare objecten die de speler doorsturen naar andere states. De muis van de speler verandert in een wijzende muis als er over een knop wordt gehoverd.

#### 3.2.1 Main Menu

Het Main Menu is het menu dat de speler krijgt te zien na de Splash State. In dit menu zijn verschillende keuzemogelijkheden te zien. Dit zijn de knoppen Start Game, Leaderboard, Settings, Quit Game en Credits. Als de speler op de knop Start Game klikt, wordt de game gestart. Leaderboard laat de vorige scores zien. Settings laat de controls van de game zien, Quit Game sluit de game af en Credits (de ronde knop met het vraagteken) laat de makers van de game zien.

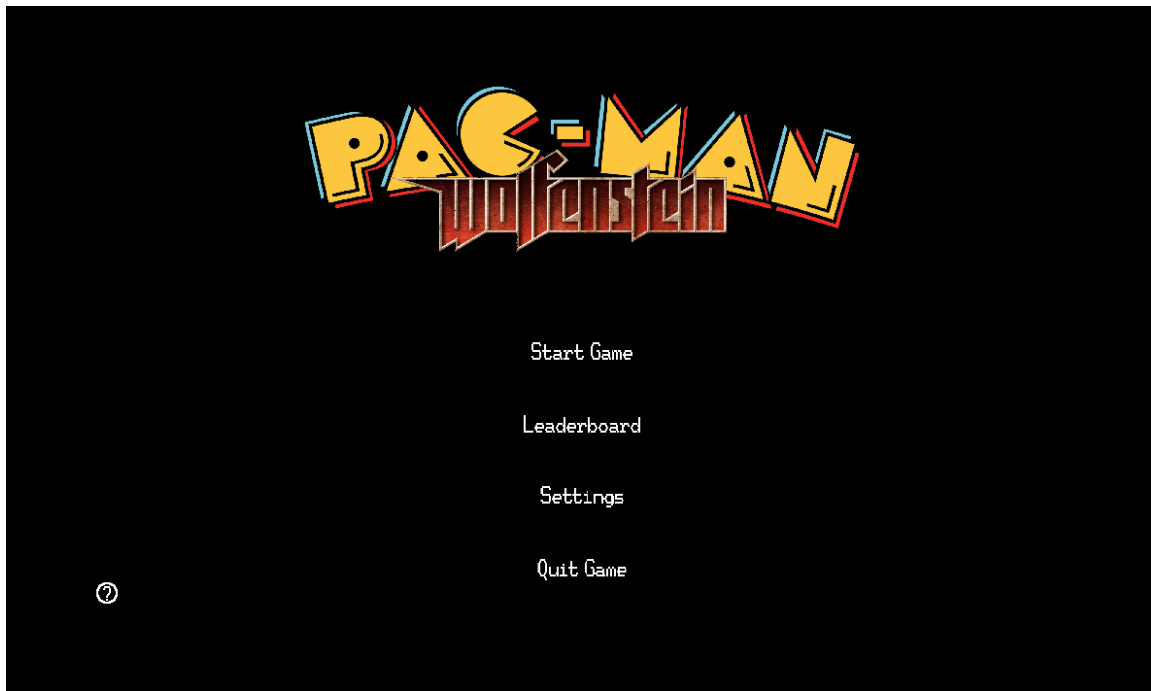


Figure 3.2: Het main menu - Begin het spel, bekijk het leaderboard, en meer.

### 3.2.2 Settings Menu

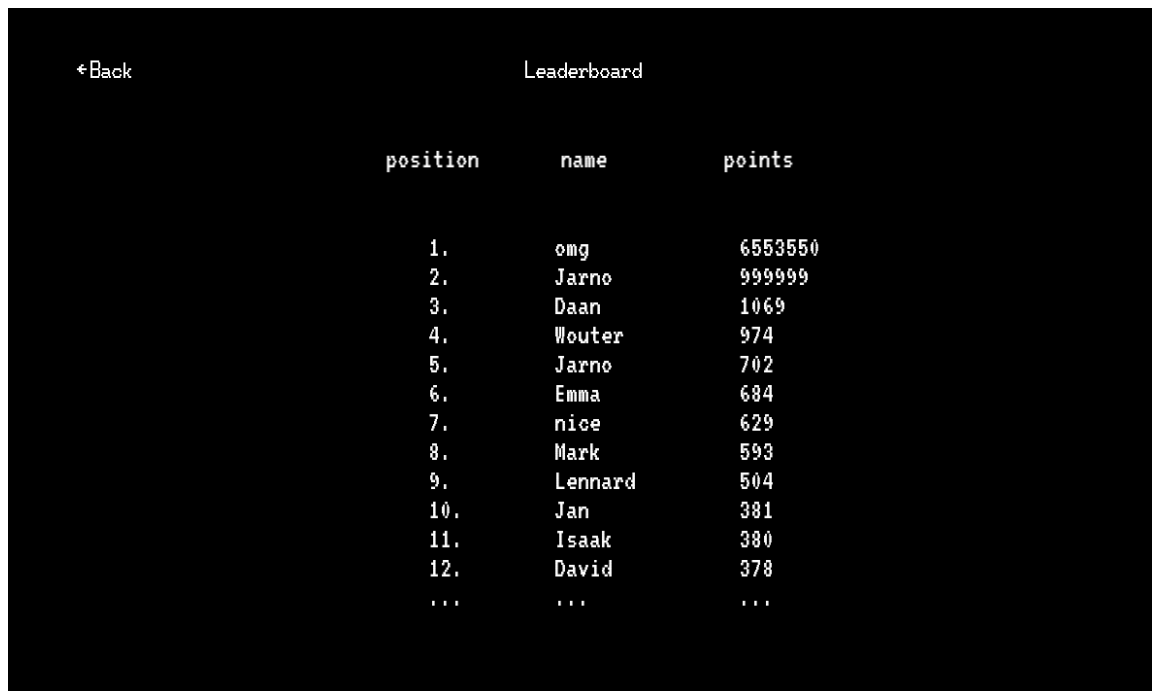
Het Settings Menu laat de verschillende controls zien die de speler kan gebruiken tijdens de game. De speler kan bewegen door middel van WASD en de pijltjestoetsen. Verder is de game te pauzeren door op de escape knop te drukken en af te sluiten door op de delete knop te drukken. De Back knop linksboven gaat terug naar het vorige menu.



Figure 3.3: Het settings menu - Hier kun je de keybindings bekijken.

### 3.2.3 Leaderboard Menu

Het Leaderboard Menu laat de scores van de voorgaande spelers zien op een aflopende volgorde. Ook is de naam van de voorgaande spelers bij de bijhorende score te zien. De Back knop linksboven gaat terug naar het hoofdmenu.



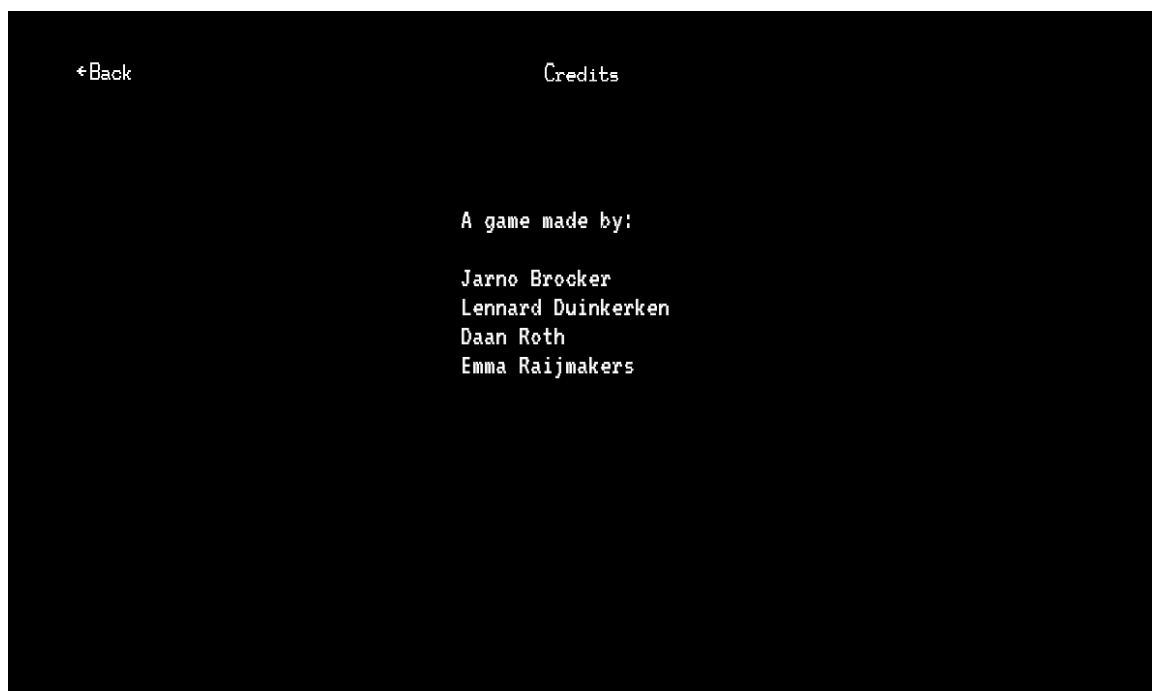
The screenshot shows a terminal window with a black background and white text. In the top left corner, there is a back button labeled '←Back'. In the top center, the title 'Leaderboard' is displayed. Below the title, there is a table with three columns: 'position', 'name', and 'points'. The table lists 12 players in descending order of points, followed by three empty rows indicated by ellipses.

position	name	points
1.	omg	6553550
2.	Jarno	999999
3.	Daan	1069
4.	Wouter	974
5.	Jarno	702
6.	Emma	684
7.	nice	629
8.	Mark	593
9.	Lennard	504
10.	Jan	381
11.	Isaak	380
12.	David	378
...	...	...

Figure 3.4: Het leaderboard - Hier staan alle scores op een rij.

### 3.2.4 Credits Menu

Het Credits Menu laat de makers van de game zien. Verder gaat de Back Button weer terug naar het Main Menu.



The screenshot shows a terminal window with a black background and white text. In the top left corner, there is a back button labeled '←Back'. In the top center, the title 'Credits' is displayed. Below the title, the text 'A game made by:' is shown, followed by a list of four names: Jarno Brocker, Lennard Duinkerken, Daan Roth, and Emma Raijmakers.

A game made by:
Jarno Brocker
Lennard Duinkerken
Daan Roth
Emma Raijmakers

Figure 3.5: De credits - Dit zijn de mensen die aan het spel gewerkt hebben.

### 3.3 In Game

De In Game State is de state waarin de gameplay begint. In deze state worden de Ghosts, Pacpellets en PowerPellets op hun posities gezet en wordt de map met alle entities door middel van Raycasting getekend. Als de speler begint met bewegen, wordt de state veranderd naar de Hunting State. De logica van de gameplay is te vinden in het hoofdstuk Game Logic.

### 3.4 Hunting

In de Hunting State kan de speler Pacpellets en Powerpellets verzamelen voor een hogere score. Pacpellets geven 10 punten Powerpellets 100 punten. Als een Powerpellet is opgepakt kan de speler de spoken opeten. Het eerste spook dat wordt opgegeten geeft 200 punten de tweede 400, de derde 800 en de laatste 1600. Een speler kan ook levens verliezen als hij aangerakt wordt door een spook. Als de speler zijn 3 levens verliest of als alle Pacpellets en Powerpellets zijn verzameld, is het spel afgelopen. De state gaat dan naar de Game Over State.

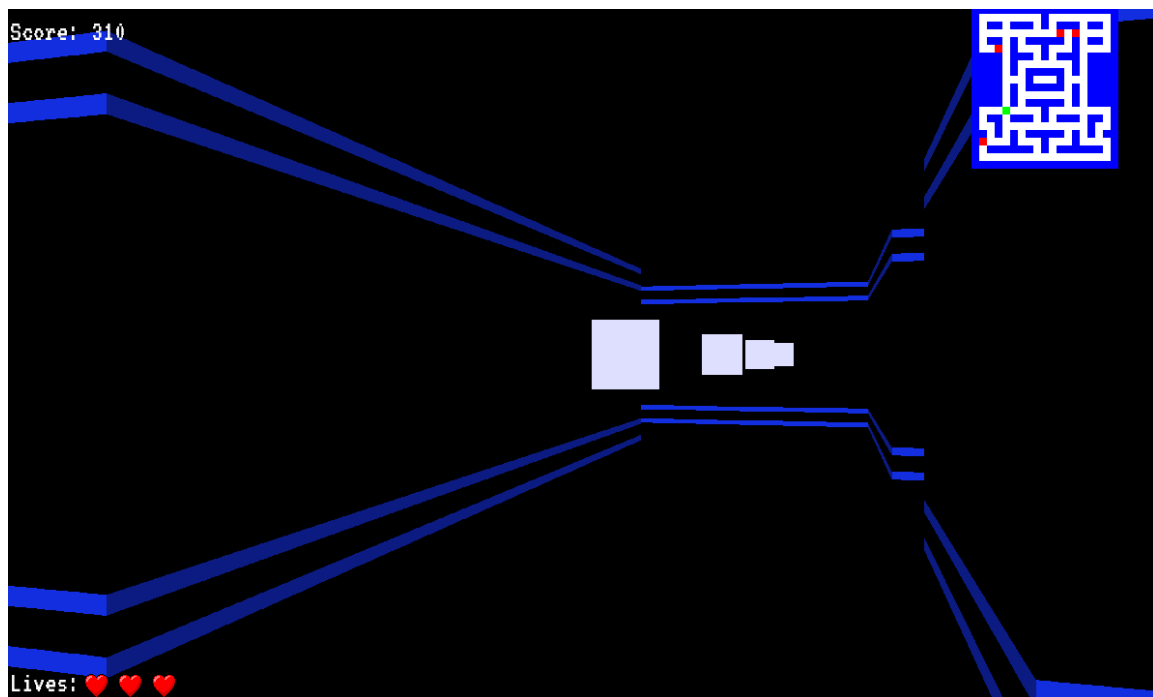


Figure 3.6: In game, hunting - Het spel is bezig en je wordt achterna gezeten.

### 3.5 Pause State

De speler kan tijdens de gameplay op Escape drukken om in de Pause State te komen. De Pause State wordt gebruikt om de game te pauzeren. In de Pause State zijn de knoppen Give Up, Continue en Settings te vinden. Give Up zorgt ervoor dat de speler naar de Game Over state gaat, Continue zorgt ervoor dat de game verder gaat en Settings gaat naar de Settings Menu State.

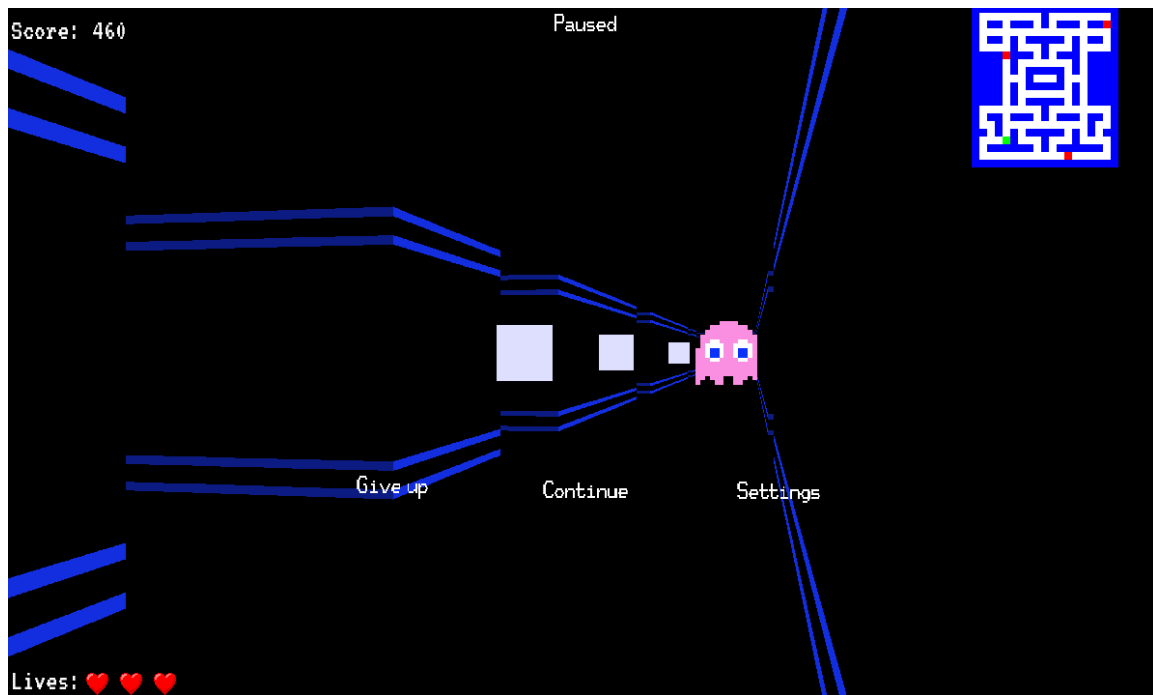


Figure 3.7: Het pauze menu - Het spel is gepauzeerd.

### 3.6 Game Over

De Game Over State is te zien als het spel is afgelopen of als de speler opgeeft. In de Game Over State is het leaderboard te zien van voorgaande scores. De score die de huidige speler heeft behaald wordt op de juiste plek in het leaderboard gezet. De speler kan vervolgens een naam invoeren bij zijn behaalde score. Door op enter te drukken wordt de score opgeslagen. Er is ook een knop, To Main Menu, om terug te gaan naar het Main Menu.



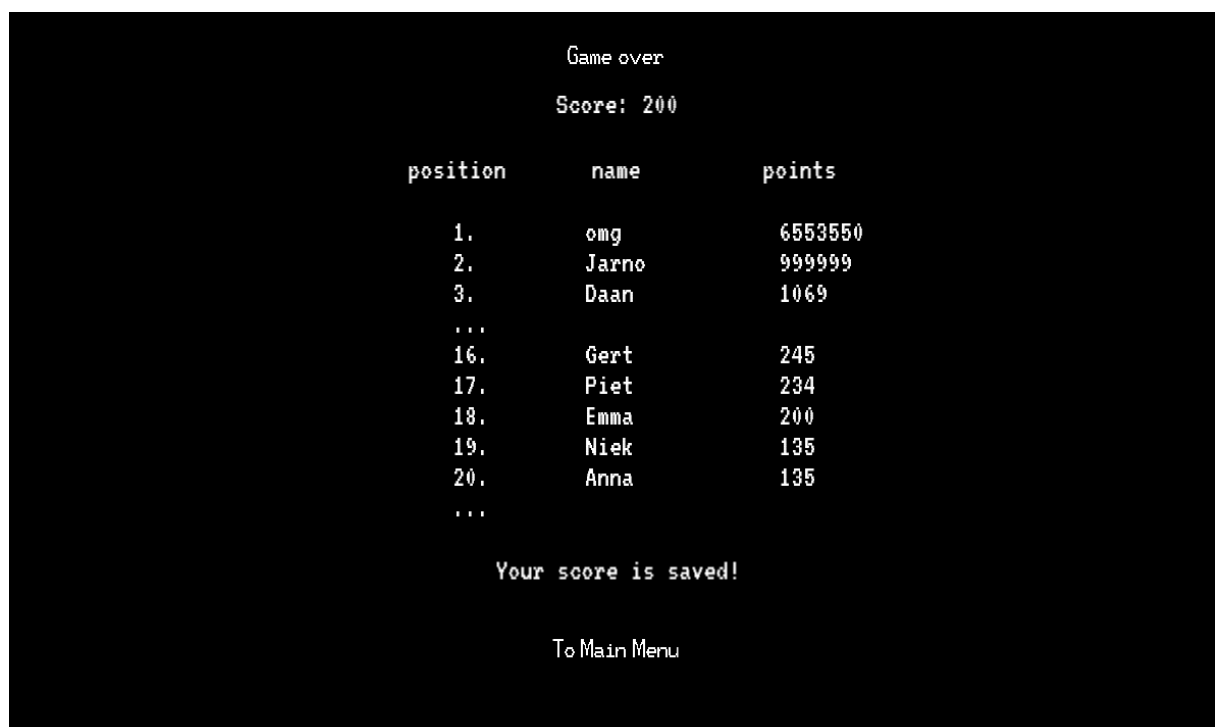


Figure 3.8: Game over - Het einde van het spel waar je je score kunt vastleggen.

## 4. Entities

Entities zijn alle objecten die gebruikt worden tijdens de gameplay. Dit zijn de Player, de Ghosts en de te verzamelen objecten: Pacpellets en Powerpellets.

### 4.1 Items

Onder items vallen alle entiteiten behalve de speler. Een Item is een object dat interactie kan hebben met de speler.

#### 4.1.1 Fruits

Fruits zijn Items die op een random positie op de map spawnen. De speler kan Fruits verzamelen voor meer punten. De Fruits zijn: Apple, Cherry, Grape, Peach en Strawberry en ieder Fruit heeft zijn eigen aantal punten.

Fruits zijn in deze versie van de game niet geïmplementeerd.

#### 4.1.2 Ghosts

Ghosts zijn de enemies van de speler. Als de speler een spook aanraakt zonder een Powerpellet te hebben gegeten, gaat er een leven van de speler af. Als de speler wel een Powerpellet heeft gegeten en binnen 10 seconden een spook opeet, gaat het spook dood en krijgt de speler punten. De ghosts zijn: Blinky (rood), Inky (cyaan), Pinky (roze) en Clyde (oranje).

### 4.2 Player

Het player object wordt bestuurd door degene die de game speelt. Door middel van WASD of de pijltjes toetsen kan de speler bewegen over de map. Een Player heeft interactie met een Item, wat de spelvariablen kan aanpassen. Zoals de score en de lives van de speler.

## 5. Resources

### 5.1 UI Elementen

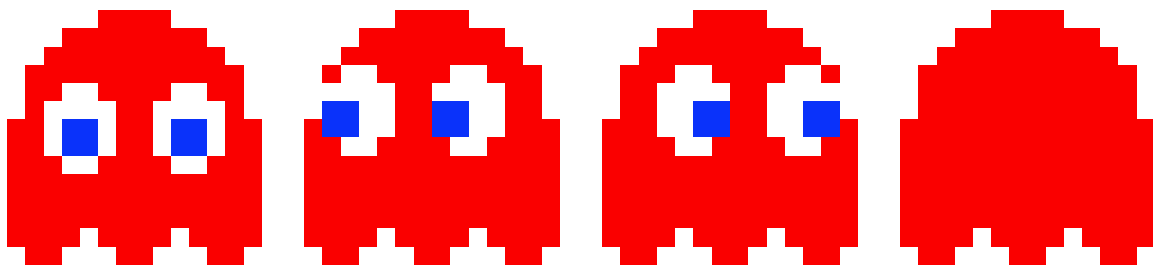
Voor de Pacenstein game zijn custom UI-elementen gemaakt. Deze elementen zijn te vinden in de UI map van de resources. De UI Elementen worden gebruikt in de menus als titels en knoppen.

### 5.2 Textures

De textures zijn afbeeldingen die gebruikt worden op de muren van de map. De texture wordt door middel van raycasting op de juiste manier getekend.

### 5.3 Sprites

De sprites zijn afbeeldingen die gebruikt worden voor de Items in de game. Dit zijn de Ghosts, Fruits en Pellets. De ghosts hebben verschillende sprites voor de verschillende richtingen waarin ze kunnen bewegen. De sprite van de ghost wordt getekend gebaseerd op de richting waarin de ghost gaat en de richting waarin de speler naar de ghost kijkt.



(a) Voorkant

(b) Rechterkant

(c) Linkerkant

(d) Achterkant

Figure 5.1: Alle hoeken van Blinky

### 5.4 Afbeeldingen

Er afbeeldingen gemaakt voor de logos van de game en de layout van de map. Deze zijn te vinden in de resource map.

### 5.5 Overige

De overige resources zijn een font, een .rc-bestand wat voor Windows beschrijft welk logo aan de executable gekoppeld moet worden, en er staan nog twee bestanden in een aparte map, data.

De bestanden in de data map zijn settings.conf en scores.csv. Settings.conf bevat de instellingen voor de game, keybindings en window parameters zoals resolutie en de optie om te kiezen voor fullscreen of niet. Het scores.csv bestand bevat de naam en bijbehorende scores.

## 6. Toekomstplannen

Wij hadden ideeën in overvloed. Dit betekend dat niet alles gerealiseerd kon worden. Een aantal dingen hadden we wel willen hebben, en met een week meer tijd was dit ook wel gelukt.

Ook zijn we een aantal dingen tegen gekomen tijdens het play-testen. Bij de demos hebben we goede ideeën gekregen van medestudenten en kregen we zelf ook een ingeving of twee zo nu en dan.

### 6.1 Ghosts Bewegen

Wij hadden het idee om het Dijkstra Algoritme toe te passen voor het lopen van de ghosts.

Dit is helaas niet gelukt omdat Dijkstra iets te complex is om naast Raycasting te implementeren in drie weken tijd. Nu lopen de ghosts een willekeurige kant op zodra ze een muur tegenkomen.

### 6.2 Fruits

Fruit toevoegen dat random op de map gespawnd wordt zodat de speler dat kan oppakken voor extra punten.

Nu hebben we alleen twee varianten van pellets. En natuurlijk de ghosts nadat je een powerpellet opgegeten hebt.

### 6.3 Map Editor

We kregen op gegeven moment het idee om een map editor te maken zodat we de bestaande map kunnen bewerken, of zodat nieuwe maps gemaakt kunnen worden.

We kwamen erachter dat het bijzonder onhandig is om telkens in een array aan te moeten geven wat er in moet komen te staan.

Dit is iets wat we tot op een zekere hoogte wel hebben; We zijn in staat om een afbeelding uit te lezen en die te gebruiken om een array automatisch te vullen. Het bewerken van de map gaat nu via Paint.

### 6.4 De Moeilijkheidsgraad

We willen nog een moeilijkheidsgraad die door de speler is in te stellen bij settings implementeren. Maar we wisten zo snel niet hoe we dit moesten doen, behalve door de punten aan te passen. Dit is niet iets wat prioriteit heeft gehad maar wat wel iets goeds zou toevoegen.

Tijdens een van de demos kregen we het idee om ook de snelheid aan te passen van de ghosts naar mate je er meer opgegeten hebt, met het idee dat het dan meer in balans blijft over de gehele duur van een spel.

Nu is het zo dat het spel tegen het einde makkelijker wordt omdat je dan ghosts opgegeten hebt, en een snelheidstweak zou dit recht kunnen zetten.

### 6.5 Keybindings

Momenteel is het niet mogelijk voor de speler om de keybindings aan te passen. Ondanks het feit dat ze settings.conf aan kunnen passen is het zo dat de keybindings niet aangepast kunnen worden. De game crashed namelijk als het wordt aangepast omdat niet voor alle mogelijke keys een afbeelding beschikbaar is.

Wat met een demo naar boven kwam is de movement snelheid van de speler. De snelheid is wat traag. Het probleem is alleen dat je met een hogere snelheid de controle over de speler kwijt raakt. Dit zou opgelost kunnen worden door een nieuwe keybinding toe te voegen; Een

voor sprinten. Sprinten zou dan betekenen dat je voor een aantal seconden een speedboost krijgt.

## **6.6 Visuele Feedback**

Iets wat tijdens de demo telkens boven kwam is dat spelers niet wisten of ze in staat waren ghosts op te eten, of wanneer ze geraakt werden door een ghost.

Dit zou opgelost kunnen worden door een overlay over het scherm te tekenen om weer te geven wat de huidige staat van de speler is.