Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Gopher Dungeon

Rapport de projet

Filière	ISC / Orientation ID
Étudiants	Elwan Mayencourt, Masami Morimura
Superviseur	Supcik Jacques
Date	12 janvier 2026
Version	1.0

Table des matières

1 Introduction

Le projet Gopher Dungeon reprend le jeu Tic-Tac-Toe et le transpose dans un environnement 3D en utilisant la technique du raycasting. Au lieu d'une grille en 2D, le joueur se déplace dans un labyrinthe en vue à la première personne, où chaque salle représente une case du plateau. Il est possible de se déplacer librement dans cet environnement et de placer son symbole (X ou O) dans la salle correspondante.

Le choix du Tic-Tac-Toe vient du fait que ses règles sont simples et bien connues, et qu'il s'agit de la proposition de base du projet. Cela permet de se concentrer principalement sur l'aspect technique, en particulier sur l'implémentation du raycasting.

L'objectif principal du projet est donc d'implémenter un moteur de raycasting fonctionnel capable de transformer une carte 2D en rendu 3D en temps réel, tout en conservant la logique du Tic-Tac-Toe. Le projet est développé en Go et compilé en WebAssembly, ce qui permet de le visualiser directement dans un navigateur web.

2 Style de code

Notre code suit des principes clairs pour garantir une bonne lisibilité et une maintenance simple sur la durée. L'objectif est de garder un code compréhensible, même lorsque la logique devient plus complexe.

- **Guard clauses** : les conditions de sortie sont placées au début des fonctions afin de limiter l'imbrication et de rendre le flux du code plus lisible
- **Variables explicites** : les noms de variables sont choisis pour décrire clairement leur rôle et leur intention
- **Fonctions à responsabilité unique** : chaque fonction effectue une seule tâche bien définie
- **Commentaires pertinents** : les commentaires sont utilisés uniquement lorsque le comportement du code n'est pas immédiatement évident

Pour assurer la qualité globale du code, nous utilisons `golangci-lint` avec une configuration stricte basée sur un Gist public bien connu (475 étoiles au 08.01.26). Le Gist est disponible à l'adresse suivante : Golden config for golangci-lint

Les principales règles activées concernent :
- la détection des erreurs non gérées (`errcheck`)
- la complexité du code (`gocognit`)
- la complexité cyclomatique (`cyclop`, `gocyclo`)
- l'utilisation de nombres magiques (`mnd`)
- l'interdiction des variables globales (`gochecknoglobals`)
- le respect des conventions du langage Go (`revive`)
- la présence et la qualité de la documentation (`godoclint`)

Cette configuration est volontairement très stricte, mais elle permet de maintenir un code propre, cohérent et plus facile à relire tout au long du projet.

3 Raycasting : Principe et fonctionnement

Le raycasting est une technique de rendu qui permet de créer l'illusion de la 3D à partir d'une carte en 2D. Le principe est simple : depuis la position du joueur, un rayon est lancé pour chaque colonne verticale de l'écran dans la direction du regard. Lorsqu'un rayon touche un mur, la distance calculée détermine la hauteur de la colonne affichée à l'écran. Plus le mur est proche, plus la colonne est haute, ce qui donne une impression de profondeur.

Cette technique était très utilisée dans des jeux comme **Wolfenstein** 3D ou **Doom**, car elle permettait de simuler un environnement 3D avec des ressources limitées. Aujourd'hui, le raycasting est remplacé par des moteurs 3D modernes, beaucoup plus efficaces mais aussi beaucoup plus complexes. Le raycasting reste par contre très intéressant, car il est assez simple à comprendre et permet de comprendre la base du rendu 3D. La figure(Fig. 1) représente un exemple de rendu en raycasting, tiré du jeu Wolfenstein 3D. image source.

Fig. 1. – « Exemple de rendu en Raycasting, Wolfenstein 3D »

Dans une scène en vue à la première personne, un paramètre important est le FOV (Field Of View), qui définit l'angle de vision du joueur. Un FOV plus large permet de voir une plus grande partie de l'environnement, mais peut aussi provoquer des déformations sur les bords de l'écran. En général, un FOV compris entre 60 et 90 degrés est utilisé pour obtenir un rendu naturel. Dans ce projet, un FOV de 1.58 radians (environ 90 degrés) a été choisi. Ce paramètre est utilisé pour calculer la direction de chaque rayon en fonction de la position du joueur et de son orientation.

Le rendu de l'image se fait en plusieurs étapes. On commence par afficher le sol et le plafond avec simplement deux rectangles de couleurs unies. Ensuite, un rayon est lancé pour chaque colonne de l'écran pour calculer la distance jusqu'au mur le plus proche. À partir de cette distance, la colonne de texture correspondante est déterminée et dessinée à l'écran. Enfin, les sprites (objets 3D) sont affichés. Ils sont triés par distance et rendus à l'aide d'un Z-buffer, qui stocke la distance de chaque colonne au mur le plus proche. Cela permet de savoir si un sprite doit être affiché devant ou derrière un mur.

3.1 Algorithme DDA (Digital Differential Analyzer)

Comme vu dans le chapitre précédent, le principe du raycasting est d'envoyer un rayon dans une direction donnée et de récupérer la distance à laquelle il touche le mur le plus proche. Cette distance est ensuite utilisée pour déterminer l'affichage à l'écran.

Une première approche consiste à avancer le long du rayon par petits incréments et à vérifier à chaque étape si un mur est touché. Cette méthode s'appelle le **Ray Marching**. Elle est simple à comprendre, mais elle devient rapidement inefficace. Elle demande beaucoup d'itérations pour

chaque rayon, surtout lorsque les murs sont éloignés. En plus de cela, choisir un incrément trop grand peut faire passer le rayon à travers certains murs.

L'algorithme **DDA** (Digital Differential Analyzer) permet de résoudre ce problème de manière beaucoup plus efficace. Au lieu d'avancer progressivement le long du rayon, il se déplace directement de cellule en cellule, en sautant d'une ligne de grille à la suivante. Cela réduit fortement le nombre de calculs nécessaires.

Les étapes principales pour chaque rayon sont :

1. **Initialisation** : calcul des distances à parcourir en X et en Y pour atteindre la prochaine ligne de grille
2. **Itération** : à chaque étape, le rayon avance vers la ligne de grille la plus proche (verticale ou horizontale)
3. **Détection** : dès qu'une cellule contenant un mur est atteinte, l'algorithme s'arrête et retourne la distance

La figure Fig. 2 montre la différence entre le ray marching et l'algorithme DDA optimisé. On voit rapidement l'avantage du DDA, qui effectue moins d'itérations pour atteindre le mur : dans cet exemple, 8 itérations pour le DDA contre 15 pour le ray marching. Cette différence peut sembler faible, mais elle devient importante lorsqu'on considère qu'un rayon est lancé pour chaque colonne de l'écran. Par exemple, pour un écran de 1280 pixels de large, cela représente 1280 rayons par frame. Avec le ray marching, cela correspondrait à environ 19200 itérations, contre seulement 10240 avec le DDA, soit presque deux fois moins de calculs.

Cet exemple reste volontairement simple, mais dans des environnements plus complexes, avec des murs éloignés, le DDA est en général beaucoup plus efficace que le ray marching.

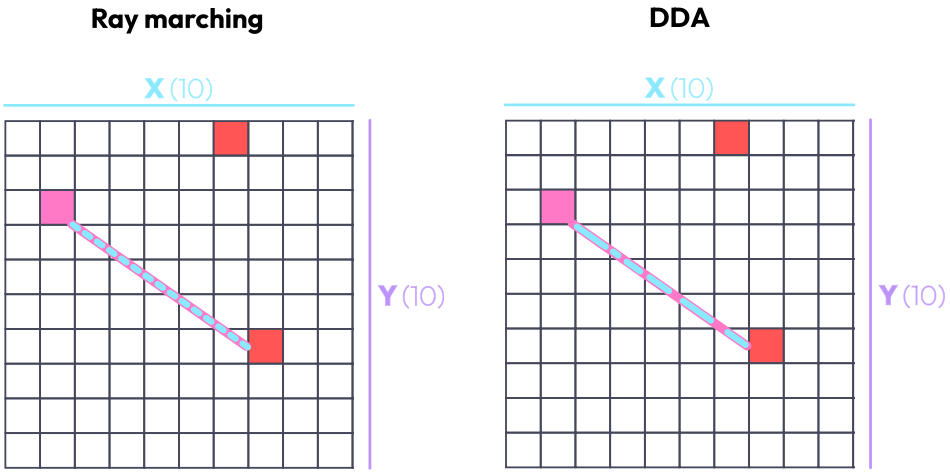

Fig. 2. – « Ray Marching vs DDA »

L'implémentation de l'algorithme DDA est guidée par les explications présentes sur le site Lode's Computer Graphics Tutorial - Raycasting.

3.2 Sélection de la texture

Dans un moteur de raycasting, il est possible d'afficher des murs avec des couleurs unies, sans utiliser de textures. Cela fonctionne, mais le rendu reste assez simple visuellement. Pour rendre

l'environnement plus intéressant et plus lisible, nous avons choisi d'ajouter des textures sur les murs.

Une fois qu'un mur est détecté par un rayon, il faut déterminer quelle partie de la texture doit être affichée. Le principe est le suivant :

1. on récupère la position exacte où le rayon touche le mur, appelée `wallX`, une valeur comprise entre 0 et 1
2. cette valeur est multipliée par la largeur de la texture pour obtenir la colonne de texture correspondante
3. cette colonne est ensuite étirée verticalement en fonction de la distance afin de créer l'effet de perspective

La figure Fig. 3 montre un rayon ainsi que les coordonnées de son point d'impact sur le mur. Selon l'orientation du mur touché (vertical ou horizontal), on utilise soit la coordonnée Y (pour un mur vertical), soit la coordonnée X (pour un mur horizontal) pour calculer la valeur de `wallX`.

Dans l'exemple présenté, le rayon touche un mur vertical. On utilise donc la coordonnée Y du point d'impact, ici 6.3. En retirant la partie entière (6), on obtient la partie décimale, soit 0.3. Cette valeur est ensuite multipliée par la largeur de la texture, par exemple 64 pixels, ce qui donne $0.3 * 64 = 19.2$. La colonne de texture utilisée est donc la colonne 19.

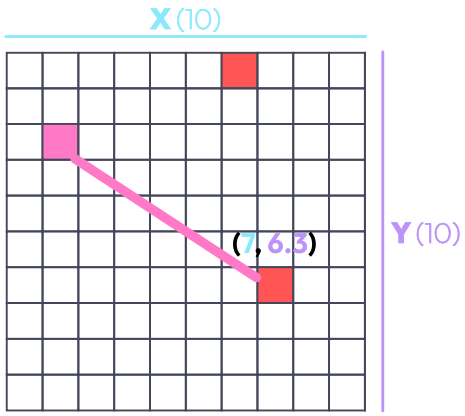

Fig. 3. – « Calcul de la coordonnée wallX pour la sélection de la texture »

4 Implémentation du moteur de Raycasting

Le moteur de raycasting est implémenté dans le fichier `raycaster.go`. Son rôle est uniquement de gérer la partie calcul : lancer des rayons, détecter les murs et calculer les informations nécessaires au rendu. L'affichage à l'écran est volontairement séparé et géré dans le fichier `world.go`. Cette séparation permet de garder un code plus clair et de distinguer nettement la logique de calcul de la logique de rendu.

Le principe est simple : pour chaque colonne de l'écran, un rayon est lancé depuis la position du joueur. Le moteur calcule la distance jusqu'au mur le plus proche, la cellule touchée dans la grille, ainsi que la position exacte de l'impact sur le mur. Toutes ces informations sont regroupées dans une structure dédiée, qui est ensuite utilisée par `world.go` pour dessiner le résultat.

La structure `RayHit` représente le résultat d'un lancer de rayon :

```
// RayHit represents the result of casting a ray in the raycasting engine.
// hit indicates if a wall was hit.
// cellX and cellY are the grid coordinates of the hit cell.
// distance is the distance from the ray origin to the hit point.
// wallX is the exact position along the wall where the ray hit (between 0 and 1).
// side indicates whether a vertical (0) or horizontal (1) wall was hit.
type RayHit struct {
  hit      bool
  cellX    int
  cellY    int
  distance float64
  wallX    float64
  side     uint8
}
```

Cette structure contient uniquement les données nécessaires au rendu. Le moteur de raycasting ne dessine rien lui-même, il se contente de retourner ces informations de manière propre et exploitable.

La fonction principale du moteur est CastRay. Elle implémente l'algorithme DDA vu précédemment. Son rôle est juste de parcourir la grille cellule par cellule jusqu'à toucher un mur ou sortir de la carte.

Cette fonction implémente l'algorithme DDA complet :

```
func CastRay(
  playerPosition Vec2,
  rayDirection Vec2,
  grid Grid,
  maxIterations int,
) RayHit {
  // grid cell the player is currently standing on
  mapX := int(playerPosition.X)
  mapY := int(playerPosition.Y)

  // deltaDistX tells us how far along the ray we must travel to cross one vertical
grid line
  // deltaDistY does the same for horizontal grid lines
  deltaDistX := math.Abs(1 / rayDirection.X)
  deltaDistY := math.Abs(1 / rayDirection.Y)

  // ...
  // setup step direction and initial side distances
  // ...

  // dda loop
  for range maxIterations {
    if sideDistX < sideDistY {
      sideDistX += deltaDistX
      mapX += stepX
      side = 0
    } else {
      sideDistY += deltaDistY
      mapY += stepY
      side = 1
    }
```

```
    if isGridCellNotEmpty(grid, mapX, mapY) {
      hit = true
      break
    }
  }

  // compute distance and wallX
  // ...

  return RayHit{
    hit:      true,
    cellX:    hitCellX,
    cellY:    hitCellY,
    distance: distance,
    wallX:    wallX,
    side:     side,
  }
}
```

Cette fonction est volontairement générique. Elle ne dépend pas du rendu, uniquement de la position du joueur, de la direction du rayon et de la grille du monde. Cela permet de tester facilement cette partie du code et de la réutiliser sans dépendre du reste du moteur.

Le fichier `world.go` s'occupe du rendu à l'écran à partir des résultats du raycasting. Il parcourt chaque colonne de l'écran et lance un rayon correspondant à cette colonne.

Pour chaque colonne, la fonction `castRayForScreenColumn` calcule la direction du rayon en fonction de la position du joueur et du FOV, puis appelle `CastRay` pour récupérer les informations de collision avec le mur. Si aucun mur n'est touché, une distance infinie est stockée dans le Z-buffer.

Lorsque le rayon touche un mur, la distance est enregistrée dans le Z-buffer. Ce tableau contient la distance du mur le plus proche pour chaque colonne et sert ensuite à gérer correctement l'affichage des sprites.

La fonction `resolveTextureStripFromHit` permet de choisir la texture et la colonne de texture à afficher en fonction de la cellule touchée et de la valeur `wallX`. La hauteur du mur projetée à l'écran est ensuite calculée à partir de la distance, puis la bande de texture correspondante est dessinée à la bonne position avec `drawTexturedWallSlice`.

```
func (w *World) raycastColumnsAndDrawWalls(screen *ebiten.Image, g *Game, p *Player)
{
  for x := range WindowSizeX {
    hit, ok := w.castRayForScreenColumn(g, p, x)
    if !ok {
      w.zBuffer[x] = math.Inf(1)
      continue
    }

    // store distance in Z-buffer
    w.zBuffer[x] = hit.distance

    strip, ok := w.resolveTextureStripFromHit(g, hit)
    if !ok {
      continue
    }
```

```
    lineH := w.wallSliceHeightOnScreen(hit.distance)
    drawStart := w.wallSliceTopY(lineH)

    w.drawTexturedWallSlice(screen, strip, x, drawStart, lineH, hit.distance)
  }
}
```

Ce découpage permet de garder une logique claire :
- `raycaster.go` s'occupe uniquement de lancer des rayons et de faire les calculs mathématiques
- `world.go` s'occupe de transformer ces résultats en pixels à l'écran

5 Gestion des textures et design

Toutes les textures du projet sont générées avec ChatGPT. Cet outil s'est révélé très pratique pour produire rapidement des textures cohérentes entre elles. Les images générées sont en haute résolution, puis redimensionnées en **64x64 pixels** dans Figma pour obtenir la taille nécessaire. Figma est également utilisé pour modifier certaines textures, par exemple pour ajouter de la surbrillance ou ajuster les couleurs.

Nous avons choisi un style **pixel art**, avec une palette plutôt sombre, pour correspondre à l'ambiance d'un donjon. Utiliser des textures de taille fixe en 64x64 pixels renforce cet effet pixelisé tout en améliorant les performances, car cela réduit la quantité de données à traiter lors du rendu.

L'ensemble du design system, incluant les textures et les choix visuels, est regroupé dans un fichier Figma accessible ici : Figma - Ray Casting Design System.

Le code ci-dessous nous permet de gérer le chargement des textures et de les préparer dans un format adapté au raycasting. La première partie, `imageManifest`, est une table de correspondance entre un `TextureID` et un nom de fichier. On l'utilise pour centraliser la liste des textures au même endroit. Cela évite d'avoir des noms de fichiers écrits en dur un peu partout dans le code, et ça simplifie l'ajout ou le remplacement d'une texture.

Ensuite, côté structures, on stocke chaque texture sous deux formes :
- Source contient l'image complète (utile si on veut afficher la texture en entier, par exemple pour le HUD)
- Strips contient la texture déjà découpée en bandes verticales de 1 pixel de large, ce qui correspond directement à la manière dont on dessine un mur en raycasting.

Un point important dans ce code est l'utilisation de l'instruction `//go:embed`. Elle permet d'inclure directement les fichiers de textures dans le binaire lors de la compilation. Cela simplifie grandement la gestion des ressources, car on n'a pas besoin de s'occuper du chargement des fichiers externes à l'exécution, tout est déjà intégré dans le programme.

Enfin, LoadTextures parcourt `imageManifest` et découpe chaque image en bandes verticales avec `sliceIntoVerticalStrips`. Le résultat est stocké dans un `TextureMap`, ce qui permet ensuite d'accéder rapidement à une texture via son TextureID. Le choix de prédécouper les textures au chargement est directement lié au rendu en raycasting. Pendant une frame, on dessine les murs colonne par colonne, donc on a besoin d'accéder très vite à la colonne de texture correspondante. Si on devait découper l'image à chaque frame, ce serait trop coûteux. Là, on fait le travail une seule fois au démarrage, puis le rendu devient simplement une sélection d'images déjà prêtes.

```
// constants.go
var imageManifest = map[TextureID]string{
  // walls
  WallBrick:       "wall-brick.png",
  WallBrickHole:   "wall-brick-hole.png",
  WallBrickGopher: "wall-brick-gopher.png",

  // sprites
  PlayerXSymbol:    "x.png",
  PlayerXCharacter: "x-player.png",
  PlayerOSymbol:    "o.png",
  PlayerOCharacter: "o-player.png",
  SkeletonSkull:    "skeleton-skull.png",
  Chains:           "chains.png",
  Light:            "lantern.png",
}

// texture.go
type TextureStrips [TextureSize]*ebiten.Image

// Texture represents a texture with its source image and vertical strips.
type Texture struct {
  Source *ebiten.Image
  Strips TextureStrips
}

// TextureMap maps texture IDs to their corresponding Texture.
type TextureMap map[TextureID]Texture

//go:embed assets/textures/*.png
var texturesFS embed.FS

func LoadTextures() (TextureMap, error) {
  for id, filename := range imageManifest {
    img, _, err := ebitenutil.NewImageFromReader(f)

    strips, err := sliceIntoVerticalStrips(img)

    out[id] = Texture{
      Source: img,
      Strips: strips,
    }
  }
}
```

6 Tests et Validation

Des tests unitaires sont mis en place pour valider les parties importantes du projet, en particulier la logique du jeu et le moteur de raycasting.

En Go, l'écriture et l'exécution des tests sont très simples. Il suffit de créer un fichier dont le nom se termine par `_test.go` et d'y définir des fonctions qui commencent par `Test...`. Go détecte automatiquement ces fichiers et exécute les tests avec la commande suivante :

```
go test -v ./...
```

Le fichier `board_test.go` se concentre sur la logique du Tic-Tac-Toe, notamment la fonction `CheckWinner`. Plusieurs cas sont testés, comme les victoires par ligne, par colonne, par diagonale et aussi les situations de match nul. Cela permet de s'assurer que l'état du jeu est correctement évalué dans tous les cas.

Pour le moteur de raycasting, le fichier `raycaster_test.go` vérifie les calculs de base :

- `TestRayDirection` s'assure que la direction du rayon est correctement calculée en fonction de l'orientation du joueur et du FOV.
- `TestRayCast` utilise une `MockGrid` pour simuler un environnement simple et vérifier que les rayons :
  - détectent correctement les murs (`hit.hit`)
  - retournent les bonnes coordonnées de cellule (`hit.cellX`, `hit.cellY`)
  - calculent une distance cohérente
  - identifient correctement le côté du mur touché (`vertical` ou `horizontal`), ce qui est important pour le rendu des textures et de l'éclairage.

7 Pipeline CI/CD

Pour éviter les erreurs et garder un projet stable, nous utilisons un pipeline CI/CD simple basé sur GitHub Actions. L'objectif est d'automatiser les vérifications du code et le déploiement, sans avoir à gérer ces étapes manuellement à chaque modification.

Deux pipelines sont utilisés :

- **CI** (`.github/workflows/ci.yml`) : vérification du code avec `golangci-lint`, suivie de l'exécution des tests unitaires à chaque push et pull request
- **CD** (`.github/workflows/cd.yml`) : compilation du projet en WebAssembly et déploiement automatique sur GitHub Pages après une CI réussie

8 Résultat final

Le résultat final est globalement très satisfaisant. L'objectif principal était de réussir à implémenter un moteur de raycasting fonctionnel et de l'intégrer à un jeu simple avec du code élégant, et nous pensons que cet objectif est atteint.

Le moteur de raycasting est stable, le rendu est fluide et l'intégration avec la logique du Tic-Tac-Toe fonctionne correctement. Le fait de pouvoir se déplacer dans un environnement 3D pour interagir avec un jeu aussi simple donne un résultat vraiment cool. Ce qui est intéressant, c'est que le code étant assez simple, il est possible de l'étendre facilement avec de nouvelles fonctionnalités.

Le jeu est jouable directement dans le navigateur à l'adresse suivante : yungbricocoop.github.io/GoTicTacToe

9 Apprentissages et améliorations

Ce projet a permis d'apprendre plusieurs aspects importants du développement logiciel, aussi bien sur le plan technique que sur le travail en équipe.

Il a notamment permis d'apprendre à collaborer sur un projet en Go. Selon le langage utilisé, la manière de structurer le code, de gérer les types et les structures peut beaucoup changer. Travailler à plusieurs sur un même codebase Go a demandé une bonne coordination, en particulier lors de modifications sur des fichiers communs. L'utilisation de Git avec des branches et des pull requests a été essentielle, même si quelques conflits sont apparus lorsque des fichiers similaires étaient modifiés en parallèle.

Sur le plan technique, le projet a permis de comprendre en profondeur le fonctionnement du raycasting et tout ce qui en découle : calcul des distances, gestion du FOV, chargement et découpage des textures, affichage des murs et rendu des sprites avec un Z-buffer. Même si le principe du raycasting est simple sur le papier, son implémentation devient rapidement complexe et désordonnée.

Le développement a également nécessité beaucoup de refactoring. La manière dont Go gère les types, les interfaces et les structures n'est pas toujours intuitive au départ, et plusieurs choix d'architecture ont évolué au fil du projet. Des fonctions ont été extraites pour rendre le code plus lisible, et le rendu a été découpé en étapes claires pour limiter la complexité.

Concernant les améliorations possibles, le code responsable de l'affichage des murs et des sprites reste encore difficile à lire et pourrait être simplifié. La qualité des tests pourrait aussi être améliorée : le coverage est relativement bas, et une meilleure séparation du code en packages faciliterait l'écriture de tests plus précis. Mais cela demanderait beaucoup de refactoring.

10 Conclusion sur Go

Go est un langage génial pour ce type de projet. Sa syntaxe est simple et lisible, ce qui permet d'avancer rapidement sans se perdre dans des abstractions complexes. La courbe d'apprentissage est plutôt douce, ce qui permet de prendre le langage en main assez vite si l'on code de temps en temps.

La cross-compilation est vraiment très pratique, surtout pour WebAssembly. Avec une simple commande `GOOS=js GOARCH=wasm go build`, le projet peut être compilé pour le web. Go fournit également `wasm_exec.js` directement dans son installation, ce qui simplifie encore le processus.

Le système de modules et la gestion des dépendances sont clairs et efficaces. L'intégration et la mise à jour de bibliothèques externes comme Ebiten se font sans difficulté.

Dans l'ensemble, Go a été une très bonne découverte pour ce projet. Go a sûrement des limitations et quelques petits défauts, mais pour un projet de cette taille et de cette complexité, il est vraiment agréable à utiliser. Nous pensons réutiliser Go pour d'autres projets à l'avenir, car c'est un langage qui fait vraiment plaisir à utiliser.

11 Références

- Lode's Computer Graphics Tutorial - Raycasting
- Lode's Computer Graphics Tutorial - Raycasting 3
- Projet en ligne
- Design System Figma