Shadow Spy
Introduction to Computer Graphics (COS426)

Simon Sure (ss9971@princeton.edu / simon.sure@inf.ethz.ch)
Flurin Steck (fs7278@princeton.edu / flurin.steck@inf.ethz.ch)

December 12, 2024

Shadow Spy is a two-player hide-and-seek game. Each player navigates a dark shared world from his own first-person perspective carrying flashlight. A player wins by finding their opponent and observe him them sufficiently long first. Players can choose between different scenes.

The game is implemented with `three.js`, decentralized WebRTC based browser-to-browser communication, and Blender 3D models with animations. We make extensive use of libraries for audio, raycasting and other features, and custom implementations collision detection and handling, first-person controls, and more.

Contents

1 Goals

Looking at past projects we decided to bring a new and unique feature to our game. We decided to make a multi-player game which influenced all subsequent design decisions.

The idea for the gameplay was quickly developed. The *Results* section elaborates extensively how the game works. We discuss more specific goals in this section.

1.1 Networking

Implementing a multi-device multiplayer game requires networked communication between the players. Most games use a centralized setup. To (a) not having to operate the server infrastructure and (b) not having to develop a separate server backend, we wanted to utilize direct browser-to-browser communication. This technology has been advanced in recent years, mostly as part of WebRTC.

Our plans for a multi-player game would require synchronization. Because the goal was to randomize the scenes, we couldn't only load static resources. During the game, the player movements would have to be synchronized at a sufficient rate for an interactive gameplay.

Using direct browser-to-browser communication generally provides lower latency than a centralized approach. This makes the game even more interactive. When both players are physically close, the latency is practically unnoticeable.

1.2 Graphics Software

We knew from the beginning that our game would require a lot of functionality to support a smooth user experience. We would have to implement first-person

controls, use ray-casting to detect when one views another user, perform collision detection, support for multiple scenes, support for networking, ... Thus we put a focus on a modular high-level software architecture for our project.

1.3 Functionality

Our initial ideas for additional features went far beyond what we had time to implement. We discussed adding weather, placing power-up items in the world, ... Given careful planing of the software architecture, those features would be reasonable to implement with a considerable time investment.

Considering the time limitation, we created a list of essential features for the game to be usable and focussed on implementing those quickly. Afterwards, we could add features as time allows.

The essential features were: First-person controls, scene bounds, score keeping, mini-map, a basic static player model, and network sync. But to have a not only technically playable but practically usable game, we had a set of very important features: Nice scenes, collision detection, and proper player model with animations.

We managed to implement all essential and utmost important features but not much more in the available time. More on additional features in the *Discussion* section.

2 Execution

We discuss the development of the project in chronological order. This section focusses on the technical aspects of the project. We discuss the non-technical aspects in sections *Goals* and *Results*.

2.1 First Person Controls

The game uses a custom first-person control implementation. Handlers are registered to run whenever a key is pressed or released. We maintain a map that indicates for keys `wsad` whether they are currently pressed. In the render loop we compute the time step since the last rendered frame and how far the player should have moved forward/back and right/left.

The player is registered as a child object of the root scene. The x and z vectors are transformed from the local player to the global coordinate system. We then move the player in those direction while adhering to scene bounds etc.

To support rotating the view, we lock the pointer. This feature makes the cursor invisible to the user. Only by pressing `ESC` is the cursor released from the browser window again. This allows us to continuously track the mouse movement.

When the mouse is moved up, the view should rotate up relative to the current view orientation, for instance.

Implementation rotations of the player has been a challenge. Initially, we represented the player as one object to which a camera was attached. Just setting the x and y rotations doesn't work. The rotations don't correspond to the intuitive coordinate frame of the current perspective. For instance, when already looking down, the y axis will be angled. Rotation around the y axis will not lead to a nice horizontal rotation.

A first approach was to compute rotation axis and perform Quaternion-based rotations around those. The final solution is more efficient and separates a head from the player body. We manage the absolute world position and rotation around the y axis at the player level and the rotation around the x axis at the head level. In the scene hierarchy the y rotation is always applied first. That corresponds to the expected and intuitive behavior of first-person controls.

2.2 Game State Machine

To manage the flow of the game, we use a global state machine. It maintains the states `SPLASHSCREEN`, `A_INIT`, `B_INIT`, `GAMEPLAY`, `SETTLING`, `WIN`, and `LOSE`. Each state is defined with `enter()`, `update()`, and `exit()` functions. The first and last are self-explanatory. `update()` is called whenever something may affect the state. The necessary actions such as a state transition or network transfers are then initiated.

When a player chooses its role as player A or player B and enters the correct game ID, a network connection is established. As soon as data can be transmitted, players progress into the respective `INIT` state.

In `A_INIT` the game is initialized. The user doesn't notice, but the browser of player A generates the scene and sets up the game for both users as soon as a scene is selected. It transmits this data to the other player and waits for a confirmation. As soon as this confirmation is received, the state transitions to `GAMEPLAY`.

In `B_INIT` the state machine waits until it receives all game information from player A. If all information is received and confirmed, it signals it readiness. When both players are ready, this player also transitions to `GAMEPLAY`.

To efficiently transmit information between both browsers, all relevant classes have custom JSON encoders and decoders. For instance, a scene is transmitted by its scene type and relevant generation seeds. The same scene can then be reconstructed on the other side. While the scene must only be synchronized once, player data is sent > 10 times per second to provide fluent motion of the other player. The data used to characterized a player are: position, orientation, score, current animation, and whether the other player is currently being observed.

In `GAMEPLAY`, each player moves through the world and computes its own

score. The own player data is frequently sent to the other player so that the latest information on the position and orientation on the opponent is always available. This information is naturally not shown to the player but used to appropriately place it in the world.

When one player reaches 1000 points, it signals the other player that the game should be ended and transitions to the `SETTLING` state. Each player then sends a message with its final state. This allows the winner to be determined. Depending on the outcome, we transition to `WIN` or `LOSE` for a result screen. At this point a new game can be started.

2.3 Global State

In addition to the game state machine, each player also maintains some globally available data. Most importantly the scene and a `GamePlay` object. Through the scene, also access to both players is possible. Having this state globally available requires rigorous management of the state. This is worth it because data about the scene and players need to be accessed in various locations to compute collisions, compute scores, update player positions from first-person controls, update the opponents information over the network, ...

2.4 Networking

As elaborated in the *Networking* part of the *Goals* section, we use a decentralized direct browser-to-browser networking approach. The underlying protocol is WebRTC and wrapped by `PeerJS`. The latter framework provides a public pairing server for WebRTC and exposes a simple interface to transmit JavaScript objects. As already mentioned, we encode/decode our objects to json instead of sending them directly. The scene and player objects are very large and cannot be serialized as a result of extending from complex `three.js` classes.

One party needs the ID of the other party and requests a connection. If the connection is established, the communication channel is available. While we only show a 4 digit code, we append a long string on both sides of the connection to that ID so that we are likely to have a unique id with the pairing server.

WebRTC is well supported by Safari and Chromium based browsers. Firefox (presumably due to data handling policies) does not have great support for WebRTC. The game should, thus, be played in Safari or Chrome.

2.5 Software Architecture

We already mentioned the state machine above. The state machine determines whether the user currently sees the splash screen, scene selection screen, wait screen, the actual gameplay, or win/lose screen. Each of those is implemented

separately. This allows very flexible extension with new features and states. The scene selection screen was introduced late during the project without issues, for example.

The configuration screens are of little interest. They add and remove html elements from the displayed html document, handle callbacks for buttons and input, and initiate the network connection.

2.6 GamePlay

The core game experience is implemented in the `GamePlay` class which relies on a `BaseScene` and `Player`. The player is a `three.js` object that is used to represent both players. Both players are constructed independent of the other each other, the scene, and gameplay object. They contain a lot of logic some of which is discussed in the subsections *Score Keeping*, *Player Model & Animations*, and *Audio*.

`BaseScene` defines our own extension of `three.js`'s `Scene`. It adds support for collision detection, variable elevation profiles, procedural generation, JSON encoding, and background audio. It does so by providing functions for each of those features. While we provide `BaseScene` as a choice to the user, it is mostly meant to be extended by more sophisticated Scene classes. `BaseScene` is used to define the interface and type between scenes in general and the `GamePlay` object.

The `BaseScene` is used as the root for rendering. It contains the attribute `world` which contains the scene content. This is to have a clear separation between the players and other utilities from the environment.

A `GamePlay` instance is created for each round. It is passed a `BaseScene` instance and two `Player` instances during construction. It sets up the first person and minimap renders. It adds the score display. It adds the players to the scene. It starts and handles first-person controls.

When the state machine decides to start the game, it calls `start()` on the `GamePlay` instance which displays the render output and kicks of the render loop. The render loop handles first-person control by tracking the key press state as elaborated above and animations as elaborated in a later subsection. As part of the render loop, it calls an update function on the own player object. This is used to transmit the player status to the opponent, update the score through ray casting, determine whether one has won, and reposition the player when both players see each other.

Developing the state machine required extensive debugging. Most issues were caused by synchronization and consistency issues between both players. We need to make sure that the shared state always agrees. We send confirmations in multiple situations to guarantee this.

In addition, we use a defensive approach. The `update()` function can be called at any time without negative side effect. Whenever a state may have changed or

something relevant might have happened, we can just call that function to handle what may have to be handled.

2.7 Minimap

The minimap in the lower right corner of the game is actually a second 3D render. It shows the camera feed of a second camera that looks down onto the game area. Other than the first-person camera this is an Orthographic Camera so that we get an undistorted top-view map. Instead of rendering the real player, we represent it through a low-quality sphere.

One could argue that rendering a second camera is 'overkill'. But because the camera is only rendering a single sphere with 16 width segments to approximate a circle, the computational effort is negligible.

2.8 Score Keeping

The player score is updated in a multi-step process:

1. Compute the distance between both players. If this distance is larger than the flashlight range we don't have to update the score.

2. Compute a direction vector that points to the other player and convert it to the local head/camera coordinate system.

   Check if the viewing direction and direction to the other player align. Alignment is determined by the flashlight cone angle.

3. If we still couldn't exclude that we observe the other player, we cast a ray through our scene and look for intersections. If no intersection in the scene is found, we observe the opponent and increase our score.

In addition to the true score value, a public score is set. This is only updated when we stop seeing our opponent. This ensures that the other player can't deduce being observed.

If one's score exceeds 1000 points, one transitions to the `SETTLING` state in the state machine to terminate the game and determine the winner.

2.9 Player Model & Animations

2.10 Scenes & Scene Selection

2.11 Collision Detection & Handling

2.12 Audio

3 Results

The final game provides a simple user experience. The game can be played by running the java script application locally or visiting https://roldsi.github.io/shadow-spy/. It is also possible for one player to use the locally deployed version and the other player to use the web-version.

To initiate a game, one player 'Player A' and the other player chooses 'Player B' on the splashscreen. Player A enters the code shown to player B. This establishes the connection between both browsers. Player A can then configure the game by choosing a scene.

- `BaseScene` is a simple plane without any objects and was originally introduced for development purposes.
- `FlowerHorror` is a flat plane that contains a few flowers at randomized positions.
- `Terrain` is a complex scene. It contains a randomized elevation profile, randomized distribution of objects such as trees, stones, etc.

As soon as player A choses a scene the game starts for both players. The user is placed at a random position in the world sufficiently far away from the other player. Depending on the browser, it may be necessary to click onto the browser window once to activate first-person controls. One can walk with keys 'wsad' and rotate by moving the cursor.

While walking the world, players can't walk through objects. Further, the light source of the flashlight is linked to the flashlight of the 3D player model. Thus, the light beam will move while walking. The flashlight has a limited range and angle. There is also background music.

On the bottom right, the player can see a mini-map of the world with its own position. Above, one can see one's own count and the other player's count. One gains points whenever one sees one's opponent. Peeking from behind an object at the opponent only counts when a sufficiently clear line of sight exists.

If one is observed by the opponent, one doesn't see the opponents score increase immediately. Their gains will only be shown once one is no longer observed to avoid using the score counter as an indicator of the presence of the opponent.

If both players observe each other, they are repositioned to new random locations in the map to avoid a stalemate.

The first player to reach 1000 points the game. As soon as that happens, the game is terminated for both players and notified whether they have won or lost. One can immediately start a new round.

4 Discussion

Any project can be improved arbitrarily. At this point, we are not aware of any big issues or bugs with the game but have various ideas for additional features or possible improvements.

We only list a selection of our ideas here to keep the report reasonably short.

4.1 Pairing & Scaling

Initially, the pairing process required that player A enters a long randomized string provided by the pairing server because every active connection with the pairing server must have a unique id. We avoid manually typing a long random string, we generate a short player code and appending a long game-specific string. This string is likely to be unique but that is not guaranteed.

If the game would be played by hundreds of people simultaneously, it will be likely that the same ID is used by multiple players. The game synchronization will break in that case.

A solution would be a more advanced pairing and discovery protocol. Considering that his is a graphics project at the core, this was out of scope.

4.2 Customization

Player A can choose from a limited set of scenes at the beginning of the game. One could add more different scenes and improve the degree of randomization within those scenes.

Additionally, one could provide the user with more customization options such as different characters to choose from.

4.3 Lighting

Given the pre-defined player model with its flashlight, there is currently limited variety in scene lighting. The game could become more interesting by adding choices between different types of flashlights: Long-range but narrow, very wide but short-range, ... Having a brighter flashlight is not necessarily an advantage because one can be tracked easier.

A great feature would also be to allow players to turn off their own flashlight. This increases the game complexity because one can try to use the other player's light to stay undetected oneself.

Besides basic flashlights, one could also introduce power-ups that can be gained throughout the game. For instance, a player could gain night vision for a limited time. This would give a one-sided advantage.

An even other option to introduce variety would be weather. Rain and fog could make visibility more difficult. While lightning could add bright momentary global illumination.

5 Ethical Evaluation

6 Resources

- Three.JS

7 Contribution

by group members