Project Documentation -- ***Int -5***

Table of Content

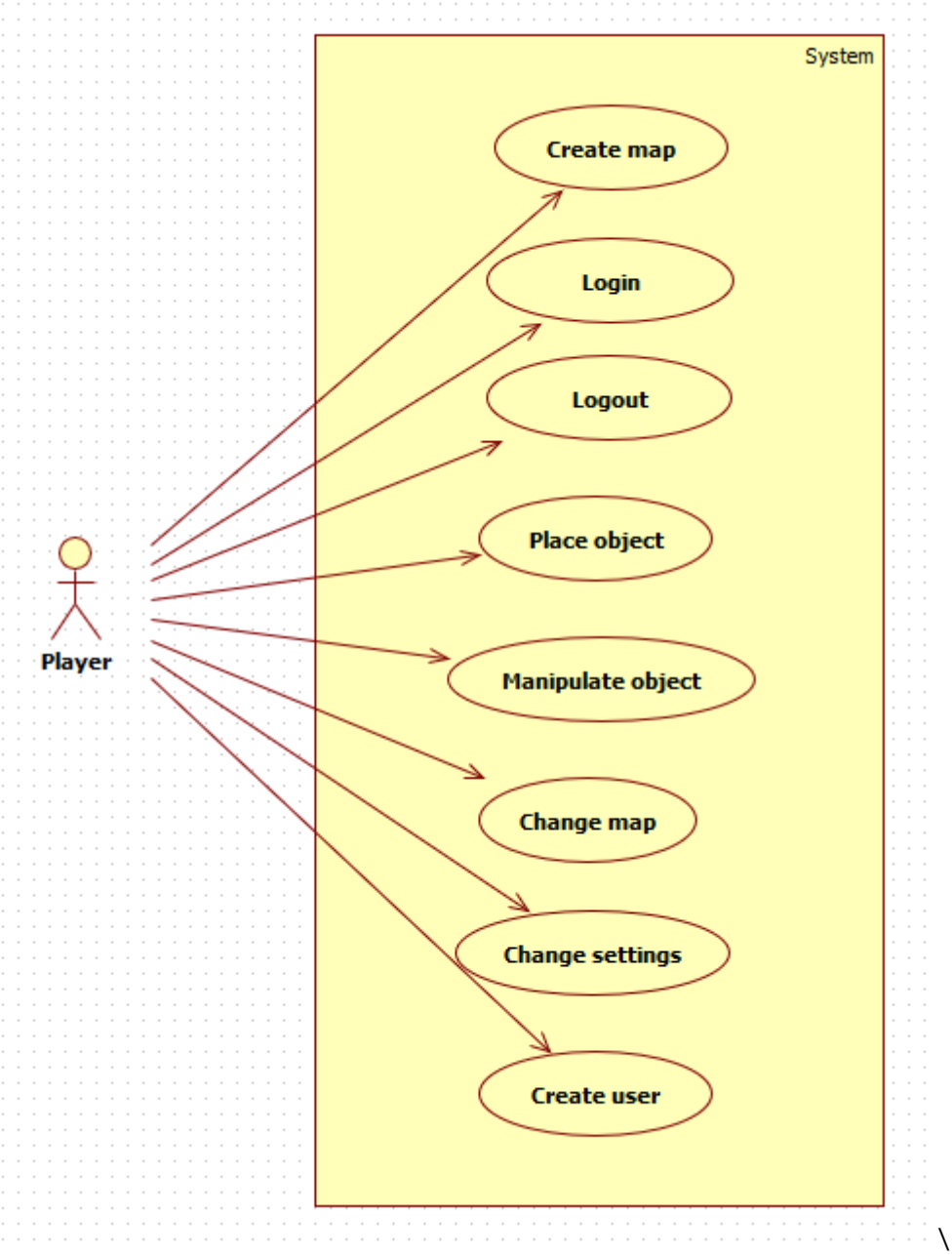
1. Introduction <Albrecht Andreas, Kaufmann Johann, Lekomzew Daniel>

1.1 Allocation of tasks

Task	**Person**
Webservice with integrated Websocket	Kaufmann Johann
User-Management	Kaufmann Johann
Unity-server connection and communication	Albrecht Andreas
Map creation and modification	Lekomzew Daniel
Object creation and modification	Lekomzew Daniel

1.2. Technical requirements

- Use-Case Diagram

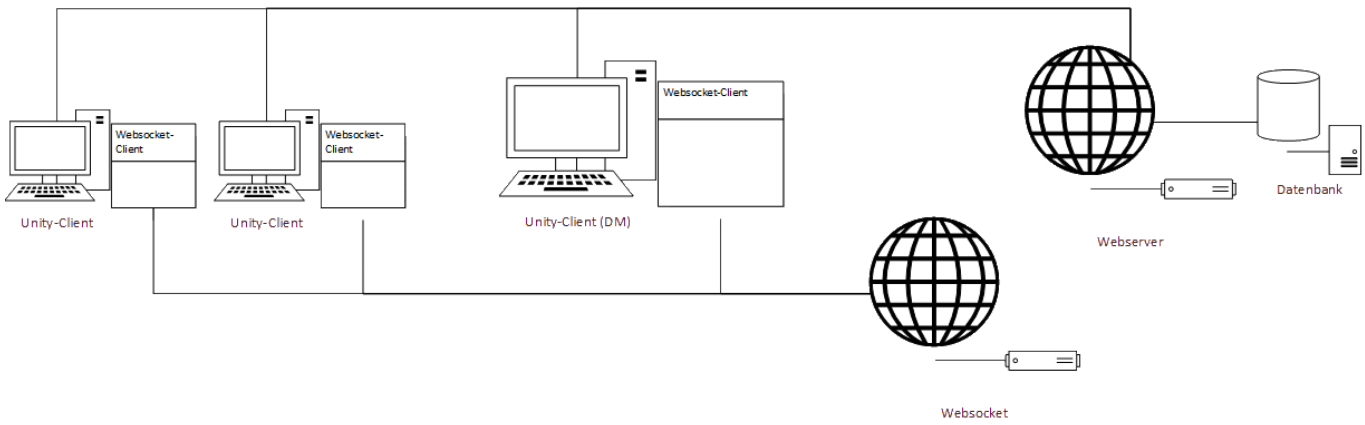


\

Picture 1 Use-Case diagram

1.3. Implementation

1.3.1. Overview Diagram



Picture 2 Overview Diagram

- The database stores user data.
- The web server reads and writes to the database.
- The web server manages currently active users, the map, and objects on the map.
- The Unity client requests and sends data from/to the web server.
- The websocket sends alerts when the web server receives new data.
- The websocket client manages communication between the websocket and Unity.

1.3.2. Sourcecode managment

Git with the WebUI GitLab was used for source code management. Due to difficulties in merging *sample_scene*, the main scene in the Unity program, only one branch was allowed to work on this scene at a time. The implementation of the requirements was done in a separate branch created for this purpose. Merge errors were handled jointly by all parties involved.

2. Subprojects

2.1. Webservice with integrated Websocket <Johann Kaufmann>

2.1.1. Is State

A Unity program is implemented with the model classes *GameObject* and *Map*. Both maps and game objects are stored client-side as fields. Consequently, all data is lost in the event of a crash or when the program is exited. The instances of the Unity program do not communicate with each other and cannot share map or game object data among themselves.

2.1.2. Purpose of the subproject

2.1.2.1 Purpose of the WebAPI

Provision of a central interface for fast and easy data transfer between clients.

2.1.2.2 Purpose of the WebsocketServer

For the connection between clients, WebSockets are utilized as an option to seamlessly implement automatic updates of both map and game object data. WebSockets were chosen due to their ability to increase network load only when necessary. Additionally, the immediate response to changes provided by WebSockets was a significant advantage. Ultimately, the decision to use WebSockets was made to prevent clients from having to fetch changes they themselves made.

2.1.3. Should State

A fully implemented web service is provided, offering HTTP GET methods for retrieving the current map and all game objects stored on the server. Additionally, HTTP POST methods are available to overwrite the stored map and add new game objects. Furthermore, an HTTP PUT method is implemented to allow modification of game objects.

Alongside the web service, there is a WebSocket server running on the same network address. Clients connect directly to the WebSocket server and are assigned a session number, which is required to utilize any of the web services. WebSocket connections are stored on the server side, and all clients, except the one making a POST/PUT request, are notified to pull the changes. On the client side, a console program manages the WebSocket connection by creating files in the temp directory, notifying the Unity program of any changes.

2.1.4. Implementation

2.1.4.1. Programming language/IDE

The IDE Visual Studio 22 was used. Its selection was based on its natural synergy with C#, the programming language used in the project. C# was chosen as the programming language due to its versatility, strong typing, and extensive framework support, making it well-suited for developing robust and efficient applications.

2.1.4.2. Used libraries + version number

- ASP.NET MVC 5.2.9
- Microsoft.Data.Sqlite

2.1.4.3. Implementation of the API

The web service was implemented as an ASP.NET web service. Simplified model classes were created for game objects and map objects, containing only the necessary fields for clients to reconstruct the objects. All interfaces are grouped within the ApiController called "DnDController," which extends the *ControllerBase* class, allowing the subclass to be mapped as a controller. All interfaces within the *DnDController* utilize the same base path of "/DnD".

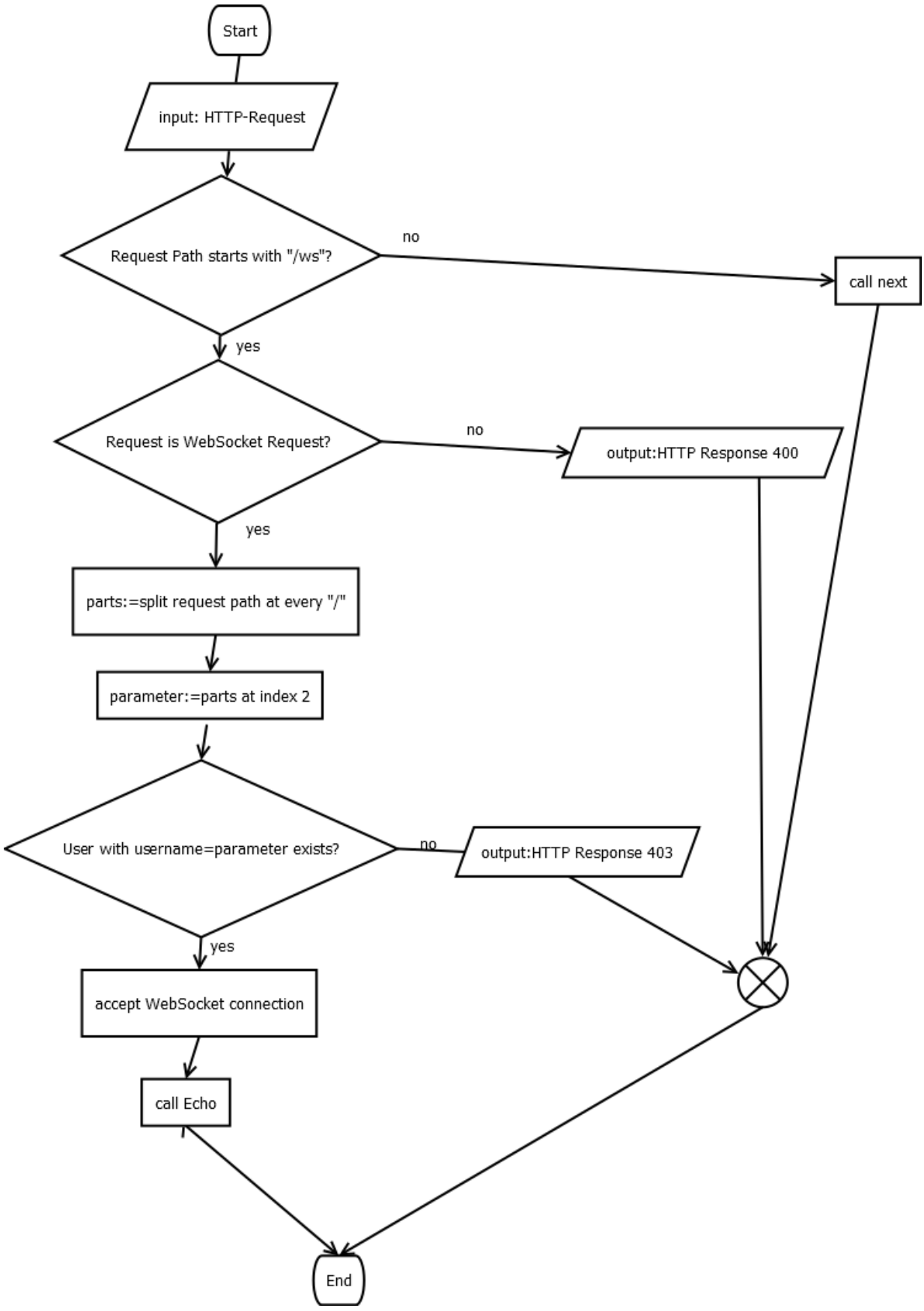
2.1.4.4. Implentation of the WebSocket

The WebSocket server was implemented as part of the existing ASP.NET Web server. The WebSocket Middleware was added using a method provided by the ASP.NET app class. The management of all connections has been delegated to the utility class called *WebSocketConnectionManager*. This class provides methods for adding and removing connections, as well as messaging all connections.

2.1.4.5. Provided Interfaces

HTTP-Method	Path	Explanation
GET	/TestConnection	returns a string if the connection to the server was built succesfully
GET	/Map/Exists	returns true if a map is saved by the server and false if not
GET	/GameObject	returns all GameObject objects currently saved by the server
POST	/GameObject	adds a new GameObject to the one saved on the server
PUT	/GameObject	changes a GameObject on saved on the server
GET	/Map	returns the currently saved map object
POST	/Map	overwrites the currently save map object

2.1.4.6. Flow Diagram



Picture 3 Flow Diagram for establishing a connection

2.1.4.7. Sample Code

2.1.4.5.7.1. PutGameObject

```
        [HttpPut]
        [ActionName("GameObject")]
        [Route("[action]")]
        public void PutGameObject([FromBody] GameObject gameObject)
        {
            if (_connectionManager.Connections.Keys.Contains(
                gameObject.ClientId))
            {
                int old = GameObjects.FindIndex(
                    g => g.Guid == gameObject.Guid);
                if (old != -1)
                {
                    GameObjects[old] = gameObject;

                }
                _connectionManager.MessageAll("ngo",
                                              gameObject.ClientId + "");


            }
            else
            {
                throw new Exception("Id not valid");
            }
        }
```

This sample code represents a HTTP PUT endpoint for updating a game object.

- The annotations "[HttpPut]", "[ActionName("GameObject")]" and "[Route("[action]")]" have been used to allow the method to be mapped by the controller mapper. These annotations define the HTTP method, the action name, and the route template for accessing this endpoint.
- The Method to be mapped, named PutGameObjekt takes a GameObject as a parameter.
- The if-clause verifies that the request was made by a client with an open connection. If the verification fails a exception with the message "Id not valid" is thrown.
- If the verification was successfull the code tries to find a GameObject with the same unique Guid as the GameObject passed as a parameter, saving it in the variable old. If it can't find a object it saves "null" into the variable instead.
- A new if-clause verifies whether the variable *old* is "null" or not. If it is not nothing is done and the method ends.
- If a object was found, it is overwritten by the passed GameObject. Then the method MessageAll from the WebSocketConnectionManager _connectionManager is called to inform all open WebSocket connections, except the one that made the request, that a new Change was made.

2.1.4.5.7.2. MessageAll

```
public async Task MessageAll(string text, string clientGuid)
        {

            foreach (var guid in Connections.Keys)
            {
                if (guid != Guid.Parse(clientGuid))
                {
                    var socket = Connections[guid].WebSocket;
                    var cTs = new CancellationTokenSource();
                    cTs.CancelAfter(TimeSpan.FromHours(2));
                    if (socket.State == WebSocketState.Open)
                    {
                        string message = text;
                        if (!string.IsNullOrEmpty(message))
                        {

                            var byteToSend = Encoding.UTF8.GetBytes(message);
                            Console.WriteLine(message);
                            await socket.SendAsync(byteToSend,
                                                   WebSocketMessageType.Text,
                                                   true, cTs.Token);
                        }
                    }
                    else
                    {
                        KillConnection(guid);
                    }
                }
            }
        }
```

The above code represents an asynchronous method named MessageAll that sends a message to all connected clients except for one identified by clientGuid.

- Within the method, it loops through the keys of a dictionary called *Connections* to iterate over the connected clients.
- For each client, it checks if the key is different from the provided *clientGuid*. If so, it retrieves the associated WebSocket object from the dictionary.
- It then checks if the WebSocket object is in an open state. If it is, the method prepares and sends the specified text message to the client using UTF-8 encoding.
- If the WebSocket is not open, it handles the termination of the connection, by calling the method *KillConnection*.

2.1.4.5.7.3. app.use + Echo

```
app.Use(async (context, next) =>
{
    if(context.Request.Path.StartsWithSegments("/ws"))
    {
        if (context.WebSockets.IsWebSocketRequest)
        {
            var parameterParts = (context.Request.Path + "").Split("/");
            var parameter = parameterParts[2];
            if (DnDController._databaseManager.UserExists(parameter))
            {
                using var webSocket = await
                    context.WebSockets.AcceptWebSocketAsync();
                await Echo(webSocket, parameter);
            }
            else
            {
                Console.WriteLine("Response 403");
                context.Response.StatusCode = StatusCodes.Status403Forbidden;
            }
        }
        else
        {
            Console.WriteLine("Response 400");
            context.Response.StatusCode = StatusCodes.Status400BadRequest;
        }
    }
    else
    {
        await next(context);
    }
});
static async Task Echo(WebSocket webSocket,string username)
{
    var buffer = new byte[1024 * 4];
    Guid guid = DnDController._connectionManager.AddUser(
        new Int5.DnD3D.WebApi.Model.User(username, webSocket));
    await webSocket.SendAsync(
        new ArraySegment<byte>(Encoding.UTF8.GetBytes(guid + "")),
        WebSocketMessageType.Text, 0, CancellationToken.None);
    DnDController._connectionManager.MessageAll("np", guid+"");
    Console.WriteLine("Neuer Client " + guid + ": " + username);
    while (webSocket.State == WebSocketState.Open)
    {
        var result = await webSocket.ReceiveAsync(
            new ArraySegment<byte>(buffer), CancellationToken.None);
        var message = Encoding.UTF8.GetString(buffer, 0, result.Count);
        Console.WriteLine(message);
    }
}
```

The above sample code is a middleware function that is executed for each HTTP request received by the application.

- It first checks if the request path starts with "/ws". If it does, it further checks if the request is a valid WebSocket request. If both conditions are met, it extracts a username , that is saved in the variable *parameter*, from the request path, checks if a user with that username exists in the database, and if so, accepts the WebSocket connection.
- If the WebSocket connection is successfully established, the Echo method is invoked asynchronously. In this method, the WebSocket connection is used to send a response to the client containing a GUID, which represents the session key needed for the consumption of the web service interfaces. It also notifies all connected WebSocket clients about the new user using the *_connectionManager* object.
- The code then enters a loop where it waits for incoming messages from the WebSocket client. It receives the messages, converts them to strings, and writes them to the console. The loop is needed, as the connection will be closed as soon as the middleware function is exited. The code within the loop however is not needed and exists merely for debugging purposes.
- If the request path does not start with "/ws", the middleware passes the request to the next middleware component in the pipeline by invoking the next delegate.

2.2. User Managment <Johann Kaufmann>

2.2.1. Is State

An ASP.NET webservice, created and described in the previous subproject.

2.2.2. Purpose of the subproject

To facilitate the representation of all currently connected users, an interface for retrieving all current users is provided. This interface allows for fetching the information of all users who are currently connected. Additionally, the ability to register new users is also implemented, enabling the registration of new users to join the system.

2.2.3. Should State

A web service is implemented to facilitate the retrieval of all open connections to the server, including the associated usernames used to establish those connections. Furthermore, a service is created for registering new users. To store user information, a lightweight database is used by the server. If no database exists on the server side, the server will create a new one.

2.2.4. Implementation

2.2.4.1. Programming language/IDE/Database

Visual Studio 2022, along with the C# programming language, was utilized for the development of the project, building upon the previous subproject. The decision to use SQLite as the database system was based on its lightweight nature, ease of creation, and the avoidance of the need for dedicated servers for each instance of the program. SQLite provides a self-contained, serverless database solution that perfectly suited the project's requirements.

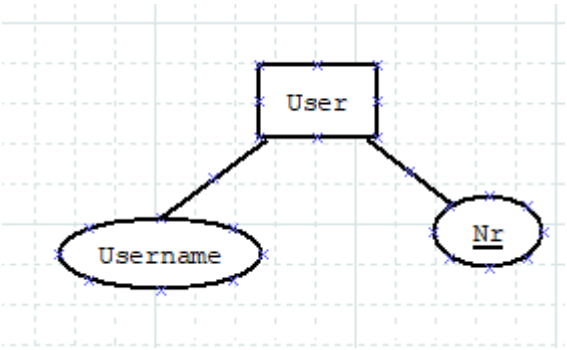
2.2.4.2. Used libraries + version number

2.2.4.3. Implementation of the User Managment

A new utility class named "DatabaseManager" has been created to handle the saving and retrieval of data from the SQLite database. Each method within the class establishes a new connection to the database, which is closed at the end of the operation to ensure proper resource management.

Following the approach described in section 2.1.4.5.7.3, the username is passed as a path parameter with the WebSocket request. To support this functionality, two new interfaces have been added to the "DnDController" class. These interfaces enable clients to add new users to the database and retrieve a list of all online users, respectively.

2.2.4.4. ER diagram



Picture 4 ER diagram of user database

2.2.4.5. Provided Interfaces

HTTP-Method	Path	Explanation
GET	/User	returns all Users that currently have a open connection
POST	/User	inserts a new User into the database

2.2.4.6. Sample Code

2.2.4.6.1. Creation of a new SQLite database

```
if (!File.Exists(@"user.db"))
{
    using (var connection = new SqliteConnection(@"Data Source=user.db"))
    {
        connection.Open();
        var command = connection.CreateCommand();
        command = connection.CreateCommand();
        command.CommandText = @"
        CREATE TABLE User (
            id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
            username TEXT NOT NULL
        );
        INSERT INTO User
        VALUES (0, 'default');
        ";
        command.ExecuteNonQuery();
    }
}
```

The above sample code represents the potential creation of a new database inside Program.cs, being run at the startup of the server.

- The code begins with a check using *File.Exists* to determine if a file named "user.db" exists in the programs working directory. If the file does not exist, the code block inside the if-statement is executed.
- Inside that code block, a SqliteConnection object is created with the connection string "Data Source=user.db" to specify the database file. The SqliteConnection is wrapped in a using statement to ensure proper resource management and automatic disposal of the connection.
- The connection is opened, and a SqliteCommand object is created. The command's *CommandText* property is set with a string containing SQL statements to create the "User" table and insert a default user with a username of "default". The table creation statement defines two columns: "id" as an integer primary key managed as an auto-increment key, and "username" as a non-null text column.
- The command.ExecuteNonQuery() method is called to execute the SQL statements. This method is used for executing non-query SQL statements like table creation and data insertion. It returns the number of rows affected, but in this case, it is not relevant.

2.2.4.6.2. AddUser

```
public bool AddUser(string username)
        {
            using (var connection = new SqliteConnection(DataSource))
            {
                connection.Open();
                var command = connection.CreateCommand();
                command.CommandText =
                    @"
                    INSERT INTO User (username) VALUES ($username);
";
                command.Parameters.AddWithValue("$username", username);
                return command.ExecuteNonQuery() == 1;
            }

        }
```

The AddUser method takes a username parameter and returns a boolean value indicating whether the user was successfully added to the database.

- Inside the method, a *SqliteConnection* object is created and opened using the *DataSource* property. It contains the connection string for the SQLite database.
- A *SqliteCommand* object is then created and assigned a SQL query using a string. The query is an INSERT INTO statement that inserts a new row into the *User* table, specifically the *username* column. The primary key *Id* of the User Table is managed as a autoincrement int. The value of the username parameter is supplied as a parameterized value to avoid SQL injection vulnerabilities.
- The *command.Parameters.AddWithValue* method is used to bind the username parameter in the query to the value of the username parameter passed to the method.
- After executing the INSERT query, the method checks the result of *command.ExecuteNonQuery()*. It returns the number of rows affected by the SQL command. If exactly one row was affected (in this case, inserted), the method returns true to indicate a successful insertion. Otherwise, it returns false.

2.3 Unity-server connection and communication <Albrecht Andreas>

2.3.1. Purpose of the subproject

The purpose of this subproject is to establish a data connection between the client and server for transporting map data, objects, and information. The WebSocket protocol is utilized to provide live updates through a background console application.

2.3.2. Implementation

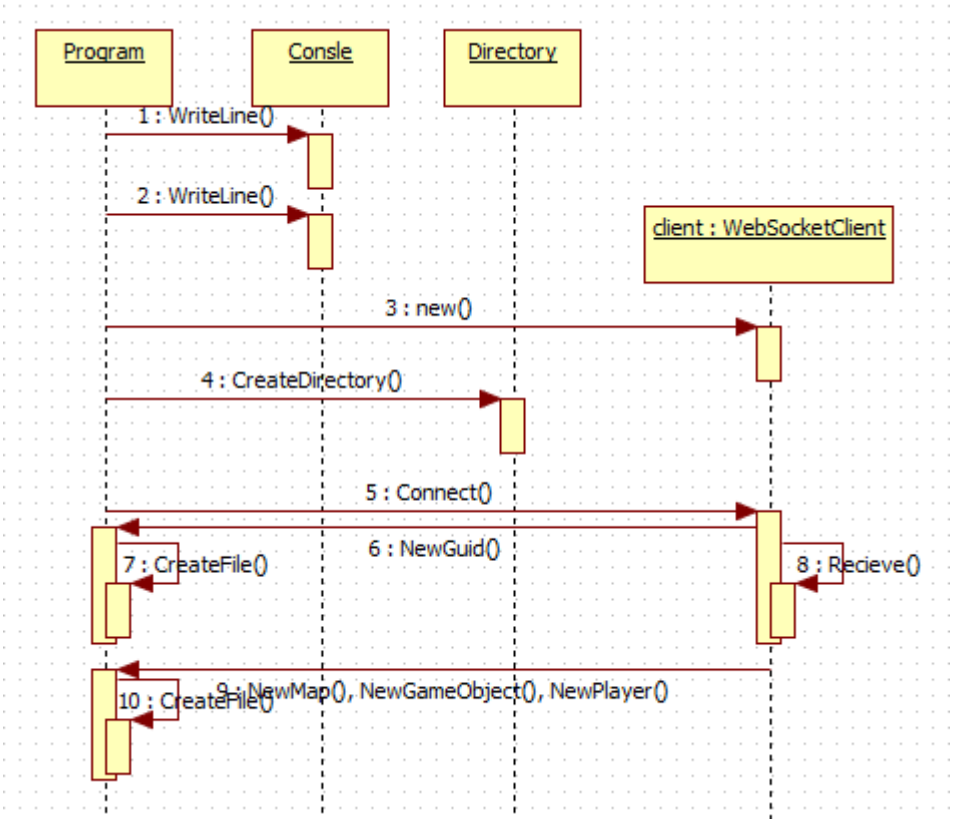
The WebSocket functionality has been implemented in a separate console application. This application writes notifications into a file. These WebSocket notifications serve as alerts to the app, indicating the type of change made without containing actual data. The Unity app awaits and reads the file. Additionally, the Unity app

directly accesses the Web service to acquire the required data. The Web request varies depending on the message received from the WebSocket.

2.3.2.1. Programming language/IDE

The programming language used for this project is C#, and the chosen IDE is Rider by JetBrains. Rider was selected for its lightweight nature, customizable features, and strong compatibility with Unity.

2.3.2.2. Diagramm



Picture 5 *Simplified Sequenzdiagram of the Websocketclient*

2.3.2.3. Used Interfaces

Websocket:

> After subscribing to the socket connection, the client receives a session ID. Subsequently, the app is automatically notified whenever the server receives any changes made by other clients, and it retrieves them from the Web service.

Webservice:

> Data is sent and requested in JSON format. The client is identified through the session ID. The HTTP protocol is used for data transfer, as the application does not require high security measures.

2.3.2.4. Provided Interfaces

Class for Interaction with the Web Service: This class provides methods to fetch and upload the required information and parse it between Unity and transfer classes.

Method	Purpose
SendMap(MapData map)	Parses a Map into MapData and then into JSON format and sends it to the Server.
AddGameObject(GameObject gameObject)	Parses a GameObject into jkGameObject format and sends it to the Server.
ChangeGameObject(Guid guid)	Retrieves the jkGameObject from an array stored in the DataContainer using the Guid as the key and sends an updated version to the server.
GameObjectsIntoDict()	The *GameObjectsIntoDict()* method is responsible for creating game objects, setting their properties (position, rotation, scale, and color), spawning them, and storing them in dictionaries for future reference.
MapExists()	Checks if the server has a map available.
FetchMap()	Retrieves the map from the server and parses it into the appropriate format for Unity.
GetUsers()	Retrieves the usernames of the currently active users.
Logout()	Safely terminates the connection to the socket.

2.3.2.5. Tests

The team conducted tests using multiple clients on one server while sharing the same map. Each client tested various changes on their own objects and at least one other object. Additionally, an independent tester performed a test.

2.3.2.6. Code Example

```
IEnumerator PostRequest(string url, string json, string method)
{
    // Debug.Log($"Sending data to {url}");
    using UnityWebRequest www = new UnityWebRequest(url, method);
    byte[] jsonToSend = new System.Text.UTF8Encoding().GetBytes(json);
    www.uploadHandler = new UploadHandlerRaw(jsonToSend);
    www.downloadHandler = new DownloadHandlerBuffer();
    www.SetRequestHeader("Content-Type", "application/json");

    yield return www.SendWebRequest();
    if (www.result != UnityWebRequest.Result.Success)
    {
        Debug.Log($"Error while Sending: {www.error} , {url}");
    }

    if (www.result == UnityWebRequest.Result.Success)
    {
        Debug.Log("Upload complete!");
    }
}
```

The above code represents the default WebRequest client from Unity. The method has a return type of *IEnumerator* because it needs to be executed as a subroutine. Due to this limitation, it cannot return values. Therefore, this method is only used for POST and PUT requests. It has not been replaced because other WebClients could only send two requests before timing out.

A *UnityWebRequest* is created, initialized with the URL to the Web service and the HTTP method.

A byte array is created and populated with a JSON string converted by the System.Text class. The *UnityWebRequest* is equipped with an *UploadHandlerRaw* that includes the byte array in the body of the HTTP request. A *DownloadHandler* is also attached to handle the response.

The SendWebRequest method returns a UnityWebRequestAsyncOperation, which can be used to create listeners for the ongoing Web request.

Finally, the success or failure of the request is logged. If the result is negative, an error message along with the URL is displayed. On a positive result, an acknowledgment message is logged.

In summary, the *PostRequest* method is used for transferring all information by sending data to the Web server.

```
public void GameObjectsIntoDict()
{
    foreach (var jkGameObject in DataContainer.Conn.GetGameObjects())
    {
        if (jkGameObject == null)
        {
            Debug.Log("jkGameObject is null");
            continue;
        }
        if (!DataContainer.GameObjects.ContainsKey(jkGameObject.Guid))
        {
            GameObject newObject =
                Instantiate(ObjectController.ModelTypes[jkGameObject.Modeltype]);
            DataContainer.GameObjects.Add(jkGameObject.Guid, newObject);
            DataContainer.Guids.Add(newObject,jkGameObject.Guid);
        }
        DataContainer.GameObjects[jkGameObject.Guid].transform.position =
            new Vector3(
            jkGameObject.pos3[0],
            jkGameObject.pos3[1],
            jkGameObject.pos3[2]
        );
        DataContainer.GameObjects[jkGameObject.Guid].transform.Rotate(
            new Vector3(
                jkGameObject.rot3[0],
                jkGameObject.rot3[1],
                jkGameObject.rot3[2]
            )
        );
        DataContainer.GameObjects[jkGameObject.Guid].transform.localScale =
            new Vector3(
            jkGameObject.scale3[0],
            jkGameObject.scale3[1],
            jkGameObject.scale3[2]
        );
        int r = Convert.ToInt32(jkGameObject.Color.Substring(0,2),16);
        int g = Convert.ToInt32(jkGameObject.Color.Substring(2,2),16);
        int b = Convert.ToInt32(jkGameObject.Color.Substring(4,2),16);


DataContainer.GameObjects[jkGameObject.Guid].transform.GetChild(0).GetChild(0)
            .GetComponent<MeshRenderer>().material.color =
            new Color(r/255f,g/255f,b/255f);
    }
}
```

The *GameObjectsIntoDict()* method is responsible for populating *DataContainer.GameObjects* dictionary with instantiated game objects.

The method loops through each *jkGameObject* in the list of game objects obtained from the webservice. Inside the loop, it checks if the current *jkGameObject* is null. If it is, it logs a debug message and continues to

the next iteration of the loop.

If the *DataContainer.GameObjects* dictionary does not already contain the Guid of the GameObject as a key, the method proceeds to instantiate a new game object based on the Modeltype stored in *ObjectController.ModelTypes*. The new instantiated object is added to the dictionary with the Guid as the key, and the corresponding Guid is added to the *DataContainer.Guids* dictionary with the new object as the key.

The method then applies the position, rotation, and scale values from jkGameObject to the instantiated game object retrieved from *DataContainer.GameObjects* using the Guid as the key. The method extracts the RGB color values from the jkGameObject string and converts them to integers.Debug messages are logged to display the Modeltype and the object's GUID.

Finally, the method accesses the child objects of the instantiated game object, retrieves the MeshRenderer component, and sets its material color based on the converted RGB values.

Overall, the *GameObjectsIntoDict()* method is responsible for creating game objects, setting their properties (position, rotation, scale, and color), spawning them, and storing them in dictionaries for future reference.

2.4 Map creation and modification <Lekomzew Daniel>

2.4.1. Purpose of the subproject

The purpose of this subproject is to enable users to create maps and modify various properties of them. The following changes can be made:

- Adjust the overall size of the map.
- Increase the height of squares on the map.
- Decrease the height of squares on the map.

2.4.2. Should State

Upon entering the map screen, the user is prompted to set the dimensions of the map if no map exists. If a map already exists, it will be loaded, and the user can modify its properties using the provided tools. These tools can be accessed from a toolbox located on the left side of the screen. Any changes made will be sent to the server and synchronized with all other clients.

2.4.3. Implementation

The map is implemented using Unity's mesh generation functionality. When the program receives information about the map, either by creating a new one or retrieving an existing one from the server, the map's mesh is calculated. The mesh comprises vertices, which represent the corner points of each square. Therefore, there are a total of (sizeX + 1) * (sizeZ + 1) vertices. Additionally, the mesh needs information about which vertices to connect in order to form the desired shape. To achieve this, we use two triangles per square, resulting in a total of [sizeX * sizeY * 6] triangle points. To apply a grid texture to the mesh, we create a material with a square texture and map the UV coordinates of the grid so it repeats itself on each square.
The "intrude" and "extrude" tools are implemented by casting a ray from the player's camera and invoking the currently selected tool at the collision point. After invoking the method, the positions of the vertices at the collision point are incremented or decremented accordingly, and the mesh is recalculated. After each modification to the map, the updated data is sent to the server using the provided methods.

2.4.3.1. Programming language/IDE

We chose to use C# as the programming language for this project, as it is the primary language used in Unity for creating the user interface. To develop the project, we utilized Rider by JetBrains as our preferred IDE, as it offers excellent Unity integration and provides tools that facilitate development tasks.

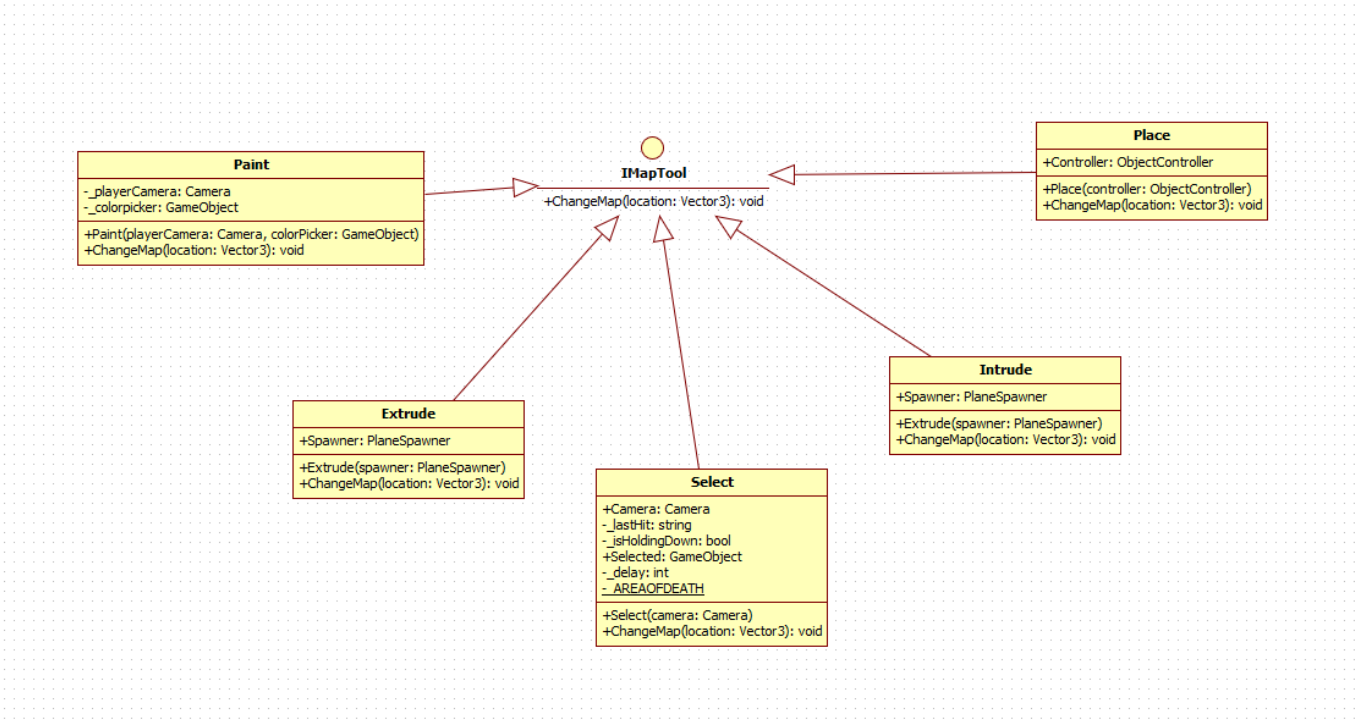
2.4.3.2. Used libraries + version number

The libraries utilized in this subproject were all components of Unity 2021.3.18f1, along with the built-in libraries of C# 6.0. No external code libraries were used. Skybox Series Free was used for the skybox.

2.4.3.3. Used Interfaces

The program utilizes an interface from the backend to transmit the updated map to the web service. This interface manages the server connection and handles the cropping of map data to eliminate unnecessary information.

2.4.3.4. Class diagrams



Picture 6 Class diagram of the maptool structure

2.4.3.5. Tests

To ensure the reliability and functionality of the program, a comprehensive testing approach was adopted. Internal team members conducted white-box testing, which involved examining the program's internal logic and components. External individuals were also involved in performing black-box testing, where the focus was on assessing the program's behavior without detailed knowledge of its internal workings. These testing methodologies were employed to identify and address any unintended behavior or potential issues within the program.

2.4.3.6. User Interface

Picture 7 Map scene user interface

- [1] Map Dimension Window: This window is automatically opened when no map is found, allowing the user to set the dimensions of the map.
- [2] Extrude Tool: Clicking on a tile with this tool selected will increment its height.
- [3] Intrude Tool: Clicking on a tile with this tool selected will decrement its height.
- [4] Button used to make [1] visible when a map already exists to set new dimensions.

2.4.3.7. Code Examples

Part of the controller for the toolbar:

```
private void Update()
{
    _delay++; // Delay is used to make it feel less sensitive
    if (Input.GetMouseButton((int)MouseButton.LeftMouse))
    {
        Ray ray = _camera.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit;
        [...]
        if (_delay < 5)
        {
            return;
        }

        _delay = 0;
        if (Physics.Raycast(ray, out hit, Mathf.Infinity))
        {
            if (hit.collider.gameObject == meshFilter.gameObject)
            {
                Vector3 mousePosition = hit.point;

                mousePosition =
meshFilter.transform.InverseTransformPoint(mousePosition);
                try
                {
                    Toolbar.MapTool.ChangeMap(mousePosition);
                }
                catch (Exception e)
                {
                    Debug.Log(e);
                }
            }
        }
    }
}
```

This code is responsible for detecting if the user has clicked on the map and applying the appropriate tool at that position. It continuously checks if the left mouse button is clicked each frame. If it is, a raycast is cast from the mouse position, and if it hits the map, the position is converted to local space. The ChangeMap() method of the currently selected tool is then called with the converted mouse position. Additionally, a delay is implemented to ensure the sensitivity of the click detection. Any exceptions that occur during the process are logged in the console for debugging purposes.

Part of the intrude tool:

```
        public PlaneSpawner Spawner { get; set; } // Contains the values for the
current map

        public void ChangeMap(Vector3 location) // location contains position
where the mouse hit
        {
            var posX = Mathf.FloorToInt(location.x);
            var posZ = Mathf.FloorToInt(location.z);
            Spawner.vertices[(posZ * (Spawner.sizeX + 1)) + posX].y += -0.5f;
            Spawner.vertices[(posZ * (Spawner.sizeX + 1)) + posX + 1].y += -0.5f;
            Spawner.vertices[((posZ + 1) * (Spawner.sizeX + 1)) + posX].y +=
-0.5f;
            Spawner.vertices[((posZ + 1) * (Spawner.sizeX + 1)) + posX + 1].y +=
-0.5f;
            Spawner.ReloadMesh();
        }
```

This code snippet represents part of the intrude tool. It is called by the tool controller (as shown in the toolbar controller code example) when an interaction with the map is detected. When the ChangeMap method of a tool is invoked, it utilizes the provided Vector3 location to modify the map at that position. In the case of the intrude tool, it identifies the bottom-left vertex of the square where the location is located and decreases the heights of all the other vertices in that square. After modifying the vertex values, the mesh is recalculated to reflect the updated values in the scene.

2.5 Object creation and modification <Lekomzew Daniel>

2.5.1. Purpose of the subproject

The purpose of this subproject is to provide the user with tools to create and modify objects on the map. The main functionalities include:

- Place Tool: Allows the user to place figures or objects on the map.
- Select Tool: Enables the user to select objects and modify their positions on different axes.
- Paint Tool: Allows the user to change the color of objects using a color picker.

These tools enhance the user's ability to create and customize their map by placing and modifying objects according to their preferences.

2.5.2. Should State

Upon loading the map screen and setting up the map, the user can utilize the place tool located on the left side of the screen. Clicking on the place tool opens a menu displaying available figures or objects. By selecting a figure from the menu, the user can then click anywhere on the map to place the selected figure. The user can repeat this process to place multiple figures until another tool is chosen.

To move the figures after they have been placed, the user can utilize the select tool. Clicking on a figure with the select tool active will display three arrows representing the X, Y, and Z axes. The user can click on an

individual arrow to move the figure along that specific axis. Alternatively, clicking on the figure itself enables the user to move it along multiple axes simultaneously.

To change the color of a figure, the user can select the paint tool. This action opens a color picker on the right side of the screen. The user can choose a desired color from the color picker. Subsequently, clicking on a figure with the paint tool active will change the color of the selected figure to the chosen color.

2.5.3. Implementation

2.5.3.1. Used libraries + version number

This subproject utilized libraries that were included in Unity 2021.3.18f1 and C# 6.0. Additionally, for the models of the Figures/Objects, freely accessible content from Thingiverse was used. The color picker functionality was implemented using judah4's HSV-Color-Picker. Further details about the models and their sources can be found at 2.5.4.

2.5.3.2. Used Interfaces

In this subproject, an interface provided by the back end of the program is utilized to send the newly created or updated objects to the web service. This interface facilitates the connection to the server and handles the cropping of the object data to remove any unnecessary information.

2.5.3.3. Implementation of object creation

The implementation of object creation and modification follows a similar approach to the extrude and intrude tool, utilizing raycasting and collision detection. When the user selects the place tool, a menu is displayed on the right side of the screen, allowing the user to choose a figure to place. These figures are stored as Unity prefabs, and upon selection, the corresponding prefab is loaded into a variable.

Once the user has selected a figure, they can click on the map to instantiate the chosen figure. The figure is instantiated at the position of the clicked location, with a value of 0 on the y-axis. The default rotation is set to Quaternion.identity.

Immediately after placing a figure, it is sent to the server using the provided methods for communication.

2.5.3.4. Implementation of object manipulation

The implementation of object manipulation allows the user to move a placed figure using the select tool. When the user clicks on a figure with the select tool active, three arrows representing the X, Y, and Z axes appear on the figure. These arrow models are stored within the figure prefab as disabled game objects and are activated when the user selects a figure.

To determine which arrow the user has clicked on, the select tool uses raycasting with collision detection. Each arrow has a tag representing its respective axis, and this tag is checked during the raycast to identify the selected arrow.

Once the user clicks on an arrow, the distance between the figure and the mouse position (converted to a world point) is calculated. The figure is then moved by that distance along the corresponding axis. If the user clicks directly on the figure instead of an arrow, the figure is moved on both the X and Z axes simultaneously.

After each update to the figure's position, the updated data is sent to the server using the provided methods for communication.

2.5.3.5. Implementation of object recoloring

The implementation of object recoloring allows the user to change the color of a figure using the paint tool. When the user selects the paint tool, a color picker menu is displayed on the side of the screen. The user can choose a color from the color picker.

After selecting a color, the user can click on a figure to change its color. This is achieved by utilizing raycasting to detect which figure the user has clicked on. The figure's material color property is then updated with the selected color.

To synchronize the color changes across all clients, the selected color is sent to the server as a hex color string. On the receiving clients, the hex color string is converted back to RGB values to update the figure's material color.

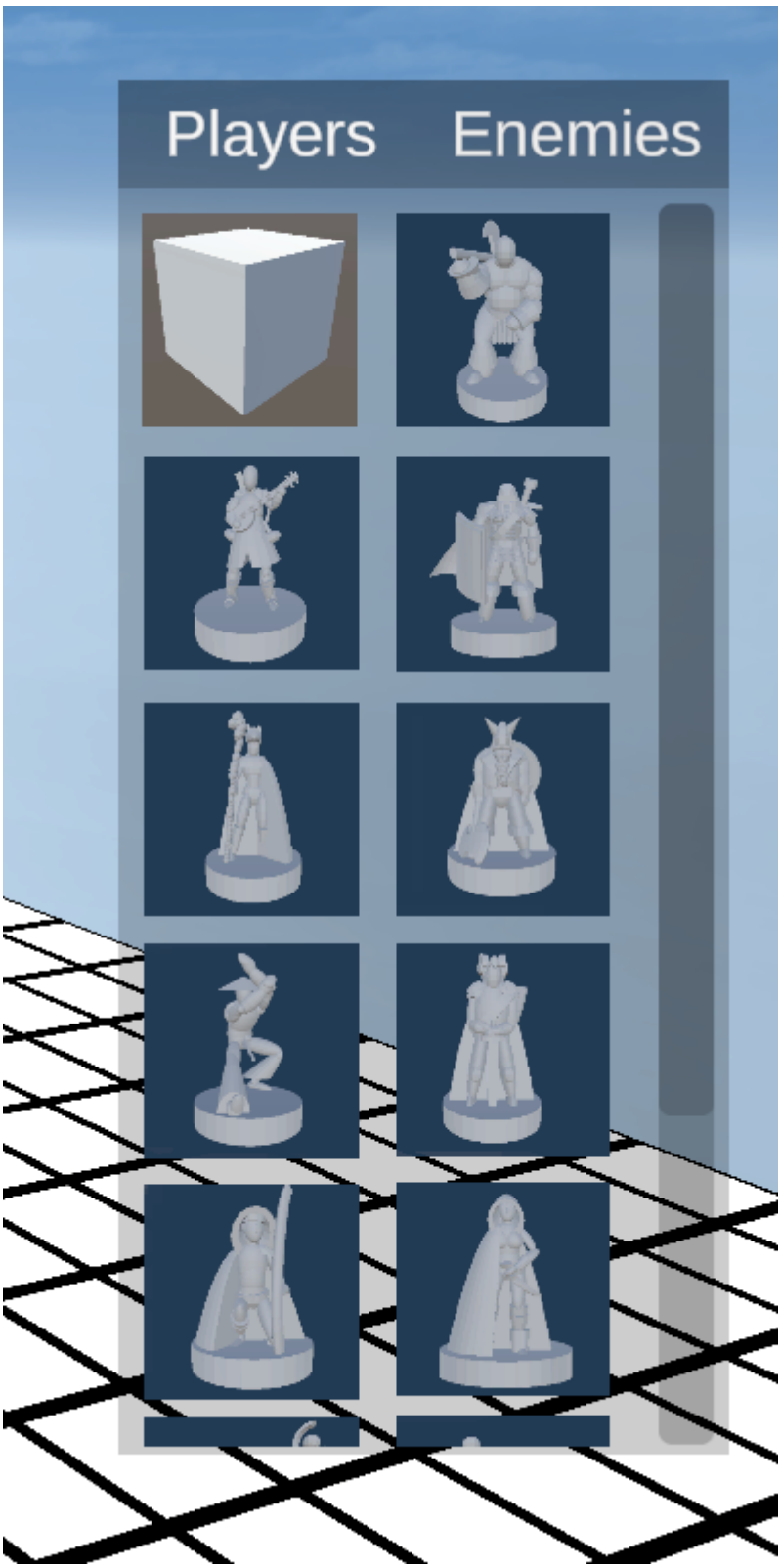
2.5.3.6. Class diagrams

See 2.4.3.4

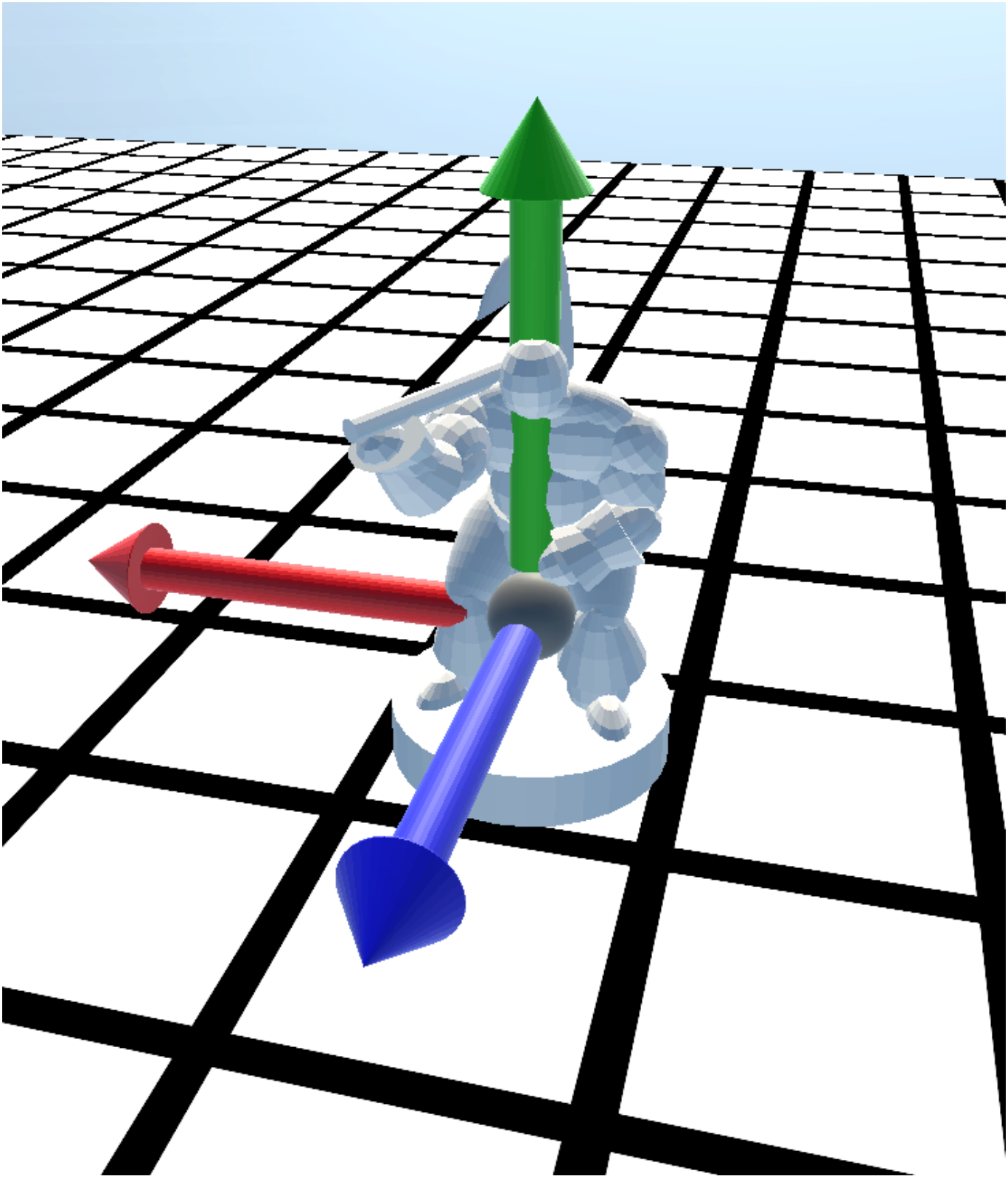
2.5.3.7. Tests

See 2.4.3.4

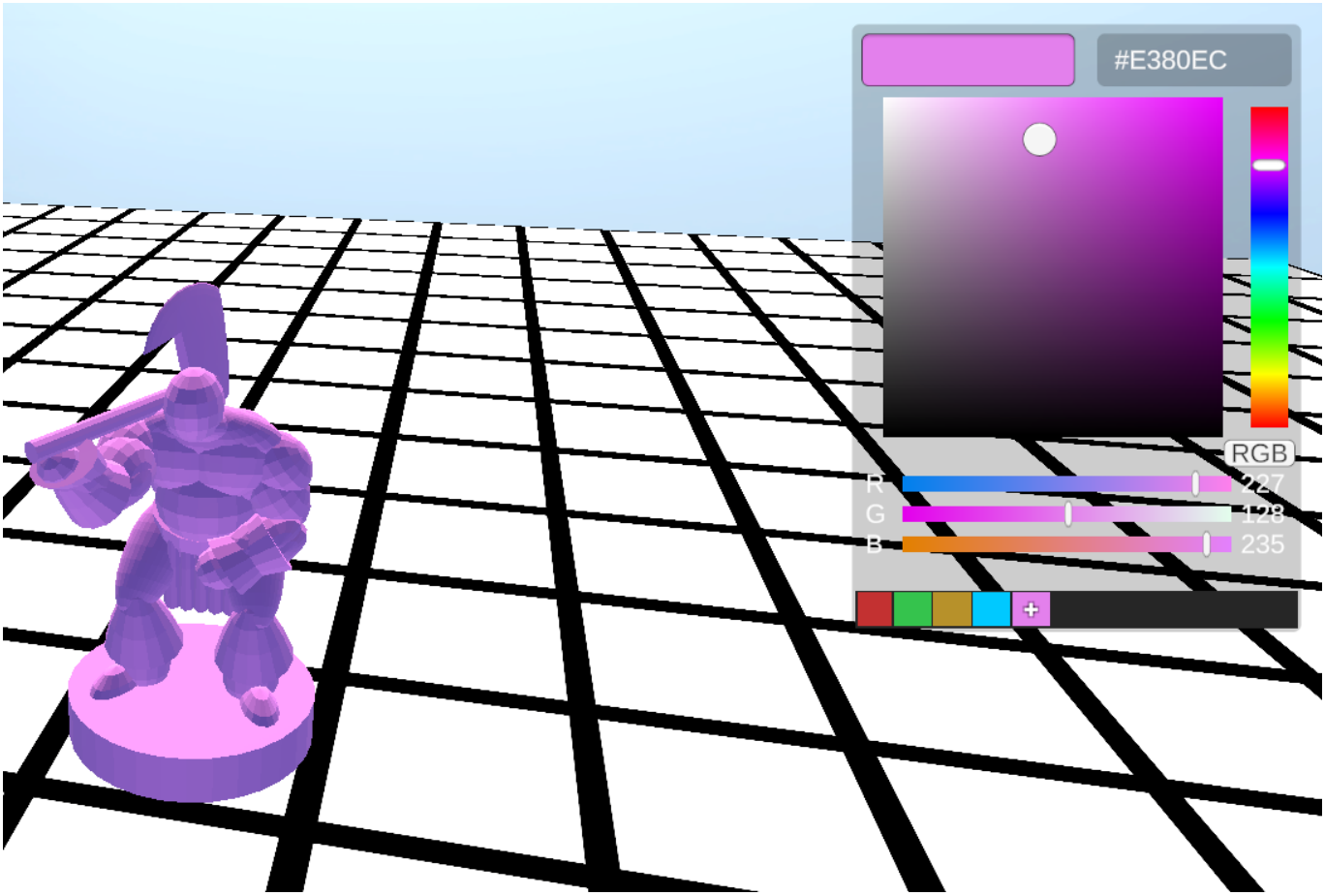
2.5.3.8. User Interface



Picture 8 Figure select menu



Picture 9 Move arrows



Picture 10 Color picker menu

2.5.3.9. Code Examples

Part of the select tool:

```
public void ChangeMap(Vector3 location)
{
    if (Input.GetMouseButtonDown((int)MouseButton.LeftMouse))
    {
        [...] // Define ray
        if (Physics.Raycast(ray, out hit, Mathf.Infinity))
        {
            if (hit.transform.CompareTag("PrefabObject"))
            {
                [...] // Deactivate on second click
                hit.transform.GetChild(1).gameObject.SetActive(true);
                Selected = hit.transform.gameObject;
            }

            if ([...]) // Check for tag
            {
                _lastHit = hit.transform.tag;
                _isHoldingDown = true;
            }
        }
    }
    else if (Input.GetMouseButton((int)MouseButton.LeftMouse) && _isHoldingDown)
    {
        [...] // Delay -> make movement less sensitive
        var position = Selected.transform.position;
        Vector3 mousePosition = Camera.ScreenToWorldPoint(new
Vector3(Input.mousePosition.x, Input.mousePosition.y, Vector3.Distance(position,
Camera.transform.position ) - 1));
        Vector3 delta = mousePosition - position;
        [...] // Area of death
        switch (_lastHit)
        {
            case "X":
                Selected.transform.Translate(Mathf.CeilToInt(delta.x), 0, 0);
                break;
            [...] // Same for Y and Z
            case "PrefabObject":
                Selected.transform.Translate(Mathf.CeilToInt(delta.x), 0,
Mathf.CeilToInt(delta.z));
                break;
        }
        [...] // Info logs
        DataContainer.Conn.ChangeGameObject(DataContainer.Guids[Selected]);
    }
}
```

The above code is responsible for enabling the user to move figures around the map. It utilizes raycasting to determine if the user has selected a figure. If an object with the "PrefabObject" tag is selected, the child object at position 1 of that object is activated. Under normal circumstances this child object is the container for the arrows used to control the figure's position. After selecting a figure, the user can click on the arrows (identified by raycasting and tags) and drag the mouse across the screen to move the figure. The code calculates the difference between the mouse position and the figure's position and then moves the figure accordingly. The movement is snapped to the grid by using Mathf.CeilToInt to round up the values. Additionally, the code sends information about the changes made to the server by using Datacontainer.Conn.

Part of the place tool

```
public void ChangeMap(Vector3 location)
{
    if (Input.GetMouseButtonDown((int) MouseButton.LeftMouse))
    {
        var posX = Mathf.FloorToInt(location.x);
        var posZ = Mathf.FloorToInt(location.z);
        GameObject instance = Instantiate(Controller.Prefab, new
Vector3(posX,0,posZ), Quaternion.identity);
        DataContainer.Conn.AddGameObject(instance);
    }
}
```

The above code is responsible for placing new figures on the map. When the left mouse button is pressed, the code uses the provided Vector3 location to determine the position where the new figure should be placed. It floors the x and z coordinates of the location to obtain integer values. Then, using the Instantiate function, a new instance of the figure is created at the specified position (with a y-coordinate of 0) using the selected prefab from Controller.Prefab. The Quaternion.identity is used for the rotation, representing no rotation. Finally, the newly created game object instance is added to the data container using DataContainer.Conn.AddGameObject(instance).

2.5.4 Sources

Name	Source
HSV-Color-Picker-Unity by judah4	https://github.com/judah4/HSV-Color-Picker-Unity
D&D Minis by Efgar	https://www.thingiverse.com/thing:945822
Beholder - 28mm D&D miniature by pyroka3d	https://www.thingiverse.com/thing:3073725
Goblin by daandruff	https://www.thingiverse.com/thing:2772094
Mimic Collection! by mz4250	https://www.thingiverse.com/thing:2843119
Minotaur - Tabletop Miniature by Yasashii	https://www.thingiverse.com/thing:3633926

3. Appendix <Albrecht Andreas, Kaufmann Johann, Lekomzew Daniel>

3.1. Glossary

L

- *Local space*

  > Object's own coordinate system relative to its position, rotation, and scale within the scene. Used for precise object manipulation, animation, and hierarchical transformations in 3D environment.

M

- *Material*

  > A set of visual and physical properties applied to objects in Unity's rendering system. It determines how light interacts with the surface, influencing its appearance and behavior. Materials define characteristics like color, texture, transparency, and reflectivity.

- *Mesh*

  > A collection of interconnected vertices that form the structural framework of a 3D object. Composed of triangles or polygons, it defines the surface and shape of the object. Essential for rendering, collision detection, and various transformations in 3D environment.

P

- *Prefab*

  > Reusable asset representing game objects with predefined properties. Facilitates efficient duplication, instantiation, and consistency across instances.

R

- *Raycast*

  > A technique for simulating light rays or detecting collisions by projecting a virtual ray into the world and determining if it intersects with objects or surfaces. It is widely used for collision detection, visibility calculations, and other interactive mechanics.

U

- *UV*

  > UV coordinates, also known as texture coordinates, are a two-dimensional mapping system applied to the vertices of a 3D model. They define how textures or images are wrapped onto the surface of the object, allowing for precise mapping and manipulation. UV coordinates are crucial for texture mapping, light baking, and creating visually detailed and realistic materials in Unity's rendering pipeline.

V

- *Vertex (Plural: Vertices)*

  > A point in 3D space that defines the position of a specific point on a mesh or model. Used in conjunction with triangles or polygons to construct the surface of a 3D object.

3.2 Sources

Name	Source
Hackmd	https://hackmd.io/
ChatGPT	https://chat.openai.com/
Skybox Series Free	https://assetstore.unity.com/packages/2d/textures-materials/sky/skybox-series-free-103633