**Team D**

**Introduction (Nikolaj Volgushev)**

ROOMS is a 2D, top-down, massively multiplayer online game. It is intended as a free-to-play application with all content in open access to the public. The main purpose of ROOMS is entertainment but also a study on the way to combine deterministic and nondeterministic game play aspects to create an innovative and involving game play experience. The main features of the game include:

* Classical MMO (massively multiplayer online) functionality: player-to-player interaction, looting and leveling
* A dynamically evolving game world consisting of an infinite number of levels, so-called rooms
* A unique algorithm for the generation of levels. The collection of special items, so-called artifacts, which enable players to directly affect the generation of levels combined with chance provide each player with a unique yet non-deterministic game experience
* High scores and level ups based on survival time
* The game website enables users to create an account, play the game, and upload and create custom content
* An editor for the creation of the custom content on the game website
* 20 different archetypes of monsters (random variations of these will be generated by the game itself), each with their own AI behavior
* Over 100 custom made items, including various weapons and supply items as well as artifacts

ROOMS can be played in the web-browser as well as on iPhone and Android mobile devices. ROOMS is developed to run in the Google Chrome web-browser and is thus playable on any personal computer OS that supports the browser. For mobile phone devices the game is supported on the iOS 6 and the Android OS. Depending on whether the game is run on a personal computer or a mobile phone, the interfaces and controls vary. Smartphone controls are implemented through the touch screen and there is only access to the core game and account creation. In the browser version of ROOMS, the in-game controls are mouse-based and users have access to all features of the website, including the editor and account management. There are three main interfaces to be considered:

* The in-game interface that is used for playing the game which includes navigation and interacting with the game world, managing the in-game inventory and communicating and interacting with other players
* The website that is used for account creation, management, starting the game and using the editor
* The editor that is used for creating, managing and uploading custom content

A mockup screen of the in-game interface is shown in Figure 1. When played in the browser, the majority of the game is controlled with the mouse. Depending on what element in the game environment the players click on, the avatar will perform a corresponding action, such as moving to a location, picking up an item or attacking a monster. Several hot keys on the keyboard can also be assigned for quick access to the inventory. Other than that the keyboard will only be used for the room chat.

All system interactions are server/client based. While most of the game logic (path-finding, monster AI, animation, etc.) occurs in the client (desktop or mobile device), the server keeps track of the current game state (currently active rooms, players stats, etc.). This leads to frequent communication between the client and the server. One of the more unique technical aspects of Rooms is that the game is an MMO playable both in the browser and on mobile devices. For that reason two main performance bottlenecks have to be taken into consideration: the server/client interaction and the limited processing power of smart phones. These are handled by limiting the amount of information sent back and forth between the client and the server to a minimum and by avoiding 3D graphics which might be too taxing to smart phone processors. There are three types of user:

* **Guests** are able to access limited features on the website, such as public rankings, viewing public user generated content, or registering to become a “Player”. These users have the most restrictive permissions, but will be allowed to play a demo version of the game anonymously.
* **Players** represent the majority of users and will be able to log into the game and access their own personal information, as well as content that they’ve generated. Players “play” the game as intended by design and can also log into the website with the same credentials to access the editor. Players may also view the player statistics of other players, including content created with the editor, and any statistics associated with it, and inherit all of the permissions of Guest users. Additionally players can manage their account through the website by managing their in-game characters and contacts.
* **Administrators** are privileged users. These users have full access to modify almost everything. For example, they can view any information available to players, ban given players from the site temporarily and/or permanently, or view currently banned players and unban them. Additionally admins have control over custom content uploaded by users.

**Background/Terminology of Rooms (Nikolaj)**

This section provides background on the ROOMS game and its various components and features to provide the reader with an understanding of MMO games, in general, and ROOMS, in particular. As shown in Figure 1, the player navigates and interacts with the game world from a 2D top-down perspective using mouse or touch screen input. The game world consists of an infinite number of levels - rooms. A room always contains doors and is filled with other content - items, monsters, containers or other players. This content is determined partly by chance and partly by the player’s current stats - primarily by artifacts but also by the player's past actions and survival time. A player may interact with the content of the room and then exit through one of the doors. The interactions include using and collecting items, fighting monsters and player-player interaction. While the combat basics are very simple-clicking on a monster deals damage-each monster archetype will have a unique AI and will require the player to try different tactics to succeed.

Upon leaving a room the player enters a new room. The old room is discarded and cannot be returned to. The new room once again is generated based on chance and the player’s stats. This process continues until either the health bar or hunger bar of the player deplete to zero or the player resigns. The objective of the game is to survive as long as possible.

One key feature of the game is the way in which the player affects the game world. This is realized by the room generation algorithm which assigns random values to a set of weighted variables. For instance, for each room there is a variable that dictates whether there is a container with items in it or not. Another variable determines what items (if any) are included in said container. Even though the assigned value of the variable is random, its weight is entirely determined by the player. This is done in three ways:

* **Artifacts**. Artifacts are found throughout the game world and are items that are carried in the player’s inventory but cannot be used actively. Their sole purpose is to affect the various weights that determine the contents of a room. For example, An artifact might increase the chances of finding supplies or increase the chances of encountering monsters. Players only have a limited amount of inventory slots so they have to chose which artifacts they want to keep and thus in which way they affect the generation of rooms. The effects of artifacts are initially unknown. The player can either determine the effects by carefully observing the changes in the environment or reveal the effects at a level up.
* **Various player actions**. Certain actions that the players perform also affect room generation. Killing a certain type of monster for example will increase the chances of that that type of monster to spawn in the next room.
* **Time passing**. The longer the player survives, the harder, and at the same time more rewarding, the game gets. Monsters and items are encountered more frequently. The monsters grow more powerful and the artifacts more useful. If a player has survived for a certain block of time a level up occurs. At level up the player can chose from five options: reveal stats of an artifact, add an inventory slot, increase health, increase hunger, get a random fact about the game world. In this way the player learns more and more about the game world and its secrets and is able to influence it too a greater extent. In the case of a player’s death, the player returns to an empty start room and is assigned a random penalty.

Another major component of the game is the multiplayer feature. Throughout the game, each player will have the potential to encounter other players who are also playing the game on their phone or computer. Whenever a player walks through a door into a new room, there is a chance that he/she will enter a room that another player is currently in (instead of randomly generating a new room). Similarly, while a player is exploring a room, at any time another player may enter it. All player to player interactions occur in real time. Players are able to communicate over a room chat, team up to travel between rooms together, trade items and attack each other.

**Operating Environment (Nikolaj)**

TThe above figure illustrates the client-server architecture of ROOMS. The client side software (website and game) run on the user's personal computer or mobile device. The client-side machine is responsible for the core of the game logic, including path-finding, the majority of monster AI and animation/ graphics. The server, which runs on a single desktop computer, is divided into two components – the game server and the web server. The game server's main task is to generate new rooms upon request from the client and track the current state of each player and room. These states are continuously communicated to the game client by the game server. The web server manages all in-game content and user data. It can both access and modify the database which stores all account and game information. For more information on the database see the information section.

The operating environment of ROOMS will be the user’s desktop (Windows, iOS), a mobile phone (Android or Apple iPhone running iOS) for the client side, and Unix-like operating system for the server side. The client side code is written in Haxe as it compiles and runs cross-platform. The server side is written in Python. The server holds all of the databases containing both dynamic user account data and static game information such as the pool of in-game items and monster archetypes. These databases are managed using MySQL. As the website includes the editor and needs to interface closely with the server, it will be implemented using Django as a web framework.



Figure 1. High-Level Architecture of Rooms

Server-to-client communication of data will primarily be done using JSON over TCP. Binary data (primarily images, but potentially other binary assets) will be base-64 encoded on the server, stored in JSON string fields, and decoded on the client. Room information will be stored in JSON objects as well, as will miscellaneous infrequently-transmitted information, such as secondary server IP addresses, client matchmaking information, and other networking details.

For client-to-server a great deal of throughput is to be expected, on the order of tens of messages a second (20-30 messages per second is common for network games). There is concern that JSON via TCP will not meet performance. The objective is to explore alternatives. One potential solution to this problem would be using UDP, and base-64 encoding the JSON data, with the intent to fit it into a single UDP packet. Another would involve using a binary networking library on the client and server side, and communicating via UDP. However, despite these drawbacks, our initial prototype will use JSON over TCP, as the simplicity in implementation is significant and it may be a premature optimization. Should this prove to be a performance issue (as is likely) we plan on experimenting with the options above.

Users are expected to have prior experience with online/offline games no lengthy gameplay tutorials will be provided. An in-game explanation of the intuitive controls will suffice for most users. The editor however will most likely be a novelty to most players and will therefore include several tutorials. While the creation of rooms features a simple drag and drop editor the creation of custom monsters will be slightly more involving. Users ambitious enough to create custom monster models will be assumed to already have some familiarity with pixelart. Therefore, the tutorial for creating custom models will focus only on the specifics of the editor and not on the creation of 2D graphics, in general.

Resource allocation will have to be adjusted as needed. Since ROOMS is an MMO, the usage pattern entirely depends on the popularity of the game and is therefore very hard to predict. One reasonable assumption is that in the starting stages of the game the user base most likely will not exceed 100 concurrent players. Since most of the more involving processing occurs on the client side a single desktop should be sufficient to uphold the server side. Another consequence of rooms being an MMO is that usage will fluctuate considerably throughout the day. Based on a statistic published by Valve on the steam-platform usage peak usually occurs around midday subsiding at night. Based on these fluctuations the server may have to adjust the amount of resources it dedicates to maintaining Rooms. However at times of high usage server/ client communication is expected to be the main bottleneck of the application. In the case that the user base of ROOMS increases past a manageable point the dedicating a faster machine to function as a server for the game might be necessary.

**Information Units (Mevludin)**

This game will require extensive use of a database to store many data objects. Specifically, ROOMS will be using MySQL as its database management system. The data stored will be user information, player/character information, monsters, items and rooms. Alongside automatically generated content, users will be able to submit custom generated content via a web-based editor. Most of the data will be persistent, with the exception of passwords. Plain text passwords will not be stored on the server, and will instead be hashed and salted. Resources such as images representing monsters, rooms, items, and players will need to be stored on the server.

 As noted above, the units of information are user information, player/character information, monsters, items and rooms. The reason for storing this information on a server is so that the user can play cross platform, and so the room generation process can use user made information from rooms, monsters and items repositories. The room, monster and item components can either be randomly generated or picked from the server. Noting that the images which represent the monsters and items will not be randomly generated, but will be stored on the server as preset data and when the user creates them using the editor. And in order for the user to access their user information from different platforms, the data will need to be stored on the server, along with their character information.

 The purpose of the user repository will be to store all of user information such as user name, real name, date of creation, hashed and salted password, and other personal information. This data is only accessible by players who are successfully authenticated. This data will not be visible to guests or other players. With a successful login,players will be able to access characters, rooms, items, and monsters that are associated with the Player. These other repositories will be explained shortly. Players will be able to edit some user information through the website and this data will only persist unless it is deleted.

 The player/character information will hold data such as the user associated to this character, health, level, location, name of player and etc. This data can only be modified by the game. Also, limited data will be available for interaction and viewing by rooms, monsters, items, and other users. This repository plays an important part of the game because without a character you cannot play the game and without a character a room cannot exist. Character information persists until a Player deletes their account or character(s).

 The monster, rooms, and items data are similar in structure because they are generated by the editor. Note that this information is not required to play the game, because this kind of information is possible to generate randomly by the game. For example, monsters health, level, attack damage, and etc, can be randomly generated. The information in these repositories will be accessed by the game at random which suits the characters current state in the game. Since this is part of the game, all characters, will be able interact with this data. For example, when the character attacks the monster, the character will be able to take away monsters health. Another example would be a random character interacting with an item. And these repositories persist until the user deletes them. Although similar in structure and purpose, their data is different.

**Performance (Thom)**

As the largest and most visible component of ROOMS is a game, and since players are unwilling to play a game which performs poorly, performance is a very important design consideration for Rooms. On the client-side, the performance requirements will vary depending upon the device. Given that a frame-rate under 24 frames per second is noticeable to the naked eye, and that the Apple Human Interface Guidelines recommend maintaining at least 30 frames per second for animated or streaming iOS Applications, the intent is to achieve over 30 FPS, even on slower hardware. Given the chosen aesthetic of ROOMS (retro, two-dimensional), we don’t believe that this should be terribly difficult to achieve, however to ensure we meet this goal, we plan on performing most, if not all network communications in a background thread, utilizing the heavily optimized Flash/NME API for image rendering, compositing, etc., and paying close attention to allocations to avoid triggering costly garbage collection.

In addition to the frame-rate requirements, ROOMS must have no noticeable loading times or loading screens. This is crucial, as many players will be interacting through the iOS or Android clients, and if there is a significant load time on these platforms, users will be inclined to close the app, and do something else (this is less of a concern on web clients, because if the loading time were egregious, a user may open a new tab, and check back. This is still clearly less than ideal, so ROOMS attempts to have no load times whatsoever). The most significant way to reduce these loading times, is to load everything in a background thread prior to when it is needed. This way, by the time a player enters the next room, it should already be either fully generated or loaded into memory (depending if the server sent a user generated room, or the rules and seed with which to deterministically generate a room on the fly). To enable this we must have a fast level generation algorithm, and all room data should be GZipped before transmission. Anything that can be reused (primarily images created by players on the website via the editor) will be cached in the client, and any pre-created assets will be bundled with the application itself.

The final client-side performance requirement we have is to minimize lag between clients, and between the client and the server. This is largely related to both the data being transmitted and the method with which its transmitted, which are both discussed in the “Operating Environment” section in some detail. We expect that our first prototype will transmit all data using JSON over TCP. This is almost certainly going to involve too much network overhead to be used for client actions (which typically get transmitted at, or slightly below framerate). The typical solution to this is to transmit data directly to other clients via UDP in real time, using a format that can fit in a single UDP Packet payload (which is around 512, after taking account headers and reassembly buffer size, less than JSON can realistically expect/guarantee), and to periodically transmit data back to the server via TCP. We expect that if we perform these steps, client-side lag will not be noticeable in the typical case.

As the server has fewer real-time guarantees to make, it has less restrictive performance requirements. The initial server should be capable of handling upwards of 100+ players simultaneously. To help enable this, we will use the [Eventlet](http://eventlet.net/) library for Python, which allows for scalable non-blocking IO. As none of the servers responsibilities are particularly strenuous on their own (which is intentional in the design), the primary performance consideration for the server will be synchronization.

 Synchronization between the client and the server is important design consideration for any multiplayer game. Typically, this is at odds with response time, as to have perfect synchronization the client would need to wait for the server’s response to execute any action. Our solution is to have clients timestamp all data they send to the server, and to not wait for the server’s response before executing actions. By transmitting a timestamp, the server and client may simulate forward with identical information, and stay synchronized with each-other within a reasonable margin of error. When simultaneous conflicts inevitably occur, the server will re-synchronize with the client, as what happened on the server is considered to be correct. This will require the server to store brief copies of past history (which should be minor compared to even image files), but has the

benefit of a low margin for error for the server and a smooth experience for players.

**Security (Nhat)**

Our main security concern is to protect user data and our game database. This includes user information and passwords, as well as user generated content. Another concern is to protect the game client and website from being exploited. This can be accomplished by restricting users from performing critical actions (such as the creation or deletion of accounts) or accessing critical information (such as passwords, private emails, etc.), from outside of the website. The intent of the system is to also ensure that users are authenticated through the server before they are allowed to play. On the website, Django’s standard user authentication system will be utilized to manage user logins. This insures user passwords are managed correctly, and additionally allows us to store boolean flags indicating actions users are allowed to and forbidden from performing. Additionally, relying on Django for security protects against SQL injection and XSS attacks, among other common web attack vectors.

In the case that ROOMS develops into a full feature game ready for a gaming platform like Steam, there are some additional security concerns to account for. However, in the case of Steam, we can use the already existing Steamworks API[1], developed by Valve Corporation. If Steam integration will be used, ROOMS would only be deployed for the desktop in production, unless Valve offers mobile integration in the future. Deploying to web browsers would be out of the question for production, so there are no security concerns there.

This API integrates features like user authentication, statistics tracking, achievements, matchmaking, and steam friends. Furthermore, ROOMS’ distribution will be handled by Valve (free of charge) along with updates. All of these features are guaranteed to be secure if we are communicating with Steam using SSL.

 Since ROOMS will be deployed to mobile platforms such as Android, iOS or even mobile web browsers, security concerns for these platforms must be considered. Since a game should not be using SSL constantly for real time data ROOMS will be using a token system. The idea is that the client will initially authenticate itself with the server for each session over SSL and will agree upon a token to use for that session. Afterwards, the client can open up a non-SSL connection and use the token until it expires. This system will allow for minimal use of SSL but will ~~also~~ provide adequate security for our application.

**Glossary and Terminology**

* Room: a level in the game. A room contains doors and other content - items, monsters, players. This content is determined at room generation.
* Room generation: The creation of a new room or the placement of the player in a currently existing room. In case a new room is created its architecture is determined randomly. A room is filled with content - monsters, items and containers. This content is determined partly by chance and partly by the player’s stats, in particular artifacts in his/her inventory.
* Monster: A computer controlled character, usually hostile towards the player. Each monster archetype has its own AI behavior and model. The player will encounter random variations of these archetypes.
* Items: Objects that the player can collect. Items divide into two subcategories - usables and artifacts.
* Artifact: a special item which affects room generation. It changes the probability of a certain type of item or monster to be placed in a room at room generation.
* Useable: Supplies to recover from hunger, regenerate health and weapons to fight monsters.
* Container: an object that contains items.
* Survival Time: The amount of time a player remains in the game without letting his health or hunger bar to deplete to zero.
* FPS: Frames Per Second, a unit describing the rate at which video frames are displayed onto a screen.
* MMO: Massively Multiplayer Online (Game), a term used to refer to multiplayer video games capable of supporting a massive number of simultaneous online users in a single game world.
* Adjacent room: Room connected to another room via a door directly.
* UDP: User Datagram Protocol, an extremely lightweight message-oriented transmission protocol which provides no guarantees about transmission reliability or transmission order. Commonly used in games for transmission of user actions or input over a network.
* TCP: Transmission Control Protocol, a reliable connection-oriented transmission protocol which guarantees the integrity and order of communicated data. Slower, but much more reliable than UDP
* GZip: A popular and freely available compression algorithm.