**p.1 Introduction** ~45s

Good afternoon everyone, thank you for attending our presentation. I’m Nathan, and with me are Max, Marcus, and Hubert. Our FYP project is Steel of Stalin, a strategy game that we built from scratch in Unity.

**p.2**

Our presentation will have 3 main sections, Overview, Implementation, and a Live demo.

**p.3**

**p.4**

So why a strategy game? A strategy game is a game where players’ actions have a high significance on determining the outcome, requiring lots of thought. As a group, we all love playing strategy games, and decided we want to make a unique strategy game of our own. Max will talk further about this in our sources of inspiration section. Since we all love strategy games, we wanted to be able to build a platform for players to express themselves through different strategies.

**p.5 - Objectives** ~45s

Going into this project, we had a few main objectives. The first of these was to make an attractive strategy game, one that is fun to play and easy on the eyes. Secondly, we wanted to develop a game where you could play with your friends. After all, all games are better with friends. And finally, we wanted to implement these goals into a smooth game experience. We wanted to have minimal lag, a pleasant experience while playing the game and overall a strong focus on user experience.

Now, I’ll give the time to Max to talk about our Sources of Inspiration

**p.6 Sources of Inspiration** ~2 mins

Thanks Nathan for introducing our project objectives. Our sources of inspiration mainly came from four famous games. They are Advanced Daisenryaku, Diplomacy, StarCraft 2, and Hearts of Iron IV.

Advanced Daisenryaku was a 2D turn-based strategy game developed by Sega in 1991. The inspiration of the command system in our project came from this game and we added even more varieties of commands in our implementation. We also adopted hexagonal tiles as the constituent of our game maps because hexagon is the most-sided regular polygon that can tessellate. Thus, it gives the most movement options to our players and it also avoids the problem of diagonal movements in square tiles because the center of a hexagon is equidistant to all of its direct neighbors.

Our next source of inspiration was Diplomacy. Diplomacy was a board game invented by Allan B. Calhamer in 1954. One of the major features of the game is that all players take their turns at the same time instead of following a sequence. This is also known as “simultaneous turns”, which can greatly reduce waiting time of all players in each round. In our project, we also adopted a similar approach for our round flow.

Our project also took a reference from Hearts of Iron IV, which is a grand strategy game developed by Paradox Interactive. The game focuses on different aspects of WWII like military, economy, diplomacy, espionage etc. The idea of customization of units came from this game as we think that it can give more variety of choices to our players. It also increases excitement by encapsulating the actual ability of the units, as the damage output is determined by the weapons that the units are wielding, which are invisible to enemies.

StarCraft2 is one of the most popular strategy games even right now. It is a real-time strategy game that allows players to execute many different strategies such as disrupting an opponent’s economy, fighting skirmishes on multiple fronts, or taking on the enemy army head on. We wanted to create the same variety of strategy in our game as well. However, due to the fast-paced nature and high mechanical intensity of real-time strategy, we decided on creating a turn-based game instead.

Next, I’ll handover to Nathan to talk about our project overview.

**p.7 - Implementation**

Now we’ll talk about our Project Implementation

**p.8 - Project Overview** ~ 45s (p8-9)

Our first and most important goal for this project was to create an “attractive strategy game”. We broke this down into four main pillars: Mechanisms, User Interface, Models, and Multiplayer. Each of these pillars contributes a large part to the attractiveness of our game.

**p.9 - Project Overview**

The Mechanisms of the game such as the game flow, map generation, and overall gameplay whether the game is fun to play. The User Interface and Models add enjoyment and visual satisfaction to the game, and the Multiplayer aspect allows a player to play with their friends.

We’ll talk more in depth about each of four core items, but first let me first introduce and clarify some of the terms we will use in the presentation.

**Definition of in-game terms**

Map: The Map is the battlefield of our game, consisting of hexagonal tiles with different types of terrain.

Terrain: There are different types of terrain, which are abstractions of real-world terrain, such as mountains, rivers, plains, forests, cities, etc.

Prop: A Prop is anything rendered on the battle scene, whether that’s a unit, a building, a tile, etc.

Unit: Units are the pieces that player’s control. They cost resources to be trained.

Round: Each round, players make decisions and issue commands to their units. This is where the player interacts with the game.

Command: A command is an action that a player can order a building or a unit under their control to perform. Some examples of commands are ordering a building to train a unit, or commanding a unit to fire on an enemy unit.

Now that I’ve introduced some of these terms, I’d like to pass the time to Marcus to talk about the Mechanisms of the game.

**p11 Game Flow**

The general game flow can be summarized as this chart. In our project, there are three scenes. At first, the main menu scene is loaded, the onstart method of the objects in the main menu scene will be triggered. And it generates several objects with backend scripts attached to handle the backend logic and multiplayer connection of the game. Then, game assets are loaded. That includes streaming assets, which are the deserialized json file which contains the information of user setting, unit data, etc. And other assets including Prefabs , background music and icons. These assets are loaded for later use. When the player starts a battle, the lobby scene will be loaded. The battle is loading behind at this point. Only after the battle is loaded and the player clicks ready, the battle scene will be loaded. After the battle ends or the player quits the battle, there is a cleanup of objects and the main menu will once again be loaded.

**p12 Round flow**

In a battle, we will have multiple rounds and the round flow acts as a gameloop to run the battle. At the beginning of the loop, the winner is checked. If there is only one surviving player, he/she will be considered as the winner and the battle will end. If not, the status of all props will be initialized or reset when a new round is started. After that, it is the planning stage for players to manage their troops and cities. This part will be handled by the UI logic. The player commands will be stored and sent to the host when the planning stage ends. The game will execute different logic, depending on whether they start as host or start as guest. We are using a multiplayer framework in both multiplayer mode and singleplayer mode. In singleplayer mode, the host logic will be executed. Bot commands are collected and executed. After all players get the result, the screen will be updated accordingly. And a new round would start.

**p13 Unit Lifecycle**

As units are the most common prop to perform all sorts of commands and actions, we would like to talk about the unit lifecycle. In this chart, the upper row is about the backend C# object and the lower row is about the Unity GameObject that represents the unit.

When a player trains a unit, it will be created, initialized and saved to the map of the battle. When the player deploys the unit, it will be added to the scene. The unit will be set as active and the corresponding prefab is also instantiated. The meshname of the unit will be set as the name of the gameObject which acts as a unique identifier to link C# object and unity gameobject. The event trigger namely PropEventTrigger will also be attached. This allows events to be handled after they are triggered. This is essential in implementing the interactions between players and the gameobjects. When the unit takes too much damage, the state of it will be set as destroyed and it will be destroyed from the scene. The corresponding backend object will be disposed at the end of the round.

Next, max will introduce the map generation algorithm.

**p14 Map Gen Intro**

Thanks Marcus for introducing the core mechanisms of our project. The implementation of the map generation algorithm contains 3 steps. First, we generate the terrains without streams and cities. Then, we generate the streams. At last, we generate the cities. Here is an example of a generated game map visualized in pixels. Green pixels represent vegetation, including grasslands, forests and jungles. Brown pixels represent highlands which include hillocks, hills and mountains. Blue pixels represent waters which include streams, rivers ,oceans and swamps. Light and dark yellow pixels are deserts and plains respectively. Red dots are the metropolises of the players and the purple dots are capturable cities.

**p15 Map Gen - Terrain**

The terrains were determined by values from 2 Perlin noise maps. One controls the height and another controls the humidity. However, we found that solely using the Unity function Mathf.PerlinNoise did not give much variation of terrains. So we added extra parameters to control the outcome of the maps generated. The frequency parameter controls how closely packed the Perlin patterns are. The octave parameter controls how detailed the patterns are. The persistent parameter controls the degree of attenuation of patterns along the octaves. The exponent parameter controls the flatness of the patterns and the warp strength parameter controls the degree of distortion of the patterns. The offset parameter controls the starting point of the generation of the Perlin maps.

**p16 Map Gen - Terrain (Cont’d)**

Here are some examples of the effect on the game map generated through varying different parameters. For easier comparison, the game map at the center uses default parameters for the height and humidity map, which are frequency, octaves, persistence and exponents are all = 1, warp strength, x and y offsets are all 0.

When the frequency is changed from 1 to 4, the Perlin patterns are much more closely packed. Notice that the lower left hand corner of the game map with frequency of 4 is in fact the original game map. When the exponent is changed from 1 to 1.2, more plains are turned into oceans because it increases the sharpness of the crests in the Perlin patterns. When the warp strength is increased to 1 from 0, the map generated is distorted a lot to give an interesting pattern. When the octaves are raised from 1 to 3, the game map generated changes drastically. If we apply 0.8 persistence to the map with octaves 3, the terrains are smoothened for a little bit. Some forest patches are gone and some grasslands are turned into plains.

**p17 Map Gen - Stream**

Next, I'll talk about stream generation. This step contains 4 sub-steps. We first pick some highlands as the water source. The number of water sources is determined by a simple ratio, which is 1 water source in 1000 highlands. Then we pick an ocean tile as the destination for each water source. Then we utilized the weighted A\* algorithm to get a path to the destination for each water source. The weights assigned for highlands are much larger than that of the flatlands to ensure that the stream generated flows from highlands to flatlands. Then we overwrite the terrains along the paths as streams, if there are any overlapped segments of the paths, convert the segments into rivers, which is a larger stream, instead.

**p18 Map Gen - Cities**

The last step of the map generation is cities generation. In order to ensure that the cities aren’t too closely packed, we took a reference from the poisson disk algorithm and modified it to suit our use. We first assign the number of cities to be generated “a”, and then we have a list of cities that are already generated. We call the count of the cities generated as “b”. If a is larger than 0, we check if b is 0 or not, if yes that means no cities have been generated yet, then we randomly pick a flatland as our first city, then minus 1 from a and add 1 to b. Alternatively if b is not 0, that means we already have at least 1 cities generated, then we read the coordinates of the last cities generated. From its coordinates, get a new coordinate by choosing a random angle of rotation and a random separation with a minimum value as the potential candidate. If the candidate is out of bounds, or not a flatland, or there’s a city already generated within the minimum separation, discard it and draw again until it is valid. Then pick the candidate as the newly generated cities, minus 1 from a and add 1 to b again, repeat until there’s no more cities to be generated.

Next. I’ll handover to Hubert to talk about the implementation of the bot algorithm.

**p19 Bot Algorithm - Bot Algorithm**

Thank you max for the map generation explanation, now I am going to talk about the bot algorithm mechanism. For the bot implementation, it is divided into 4 main parts which are Training, Construction, Movement, and Combat, the bot algorithm flow will go as the following graph.

**p20 Bot Algorithm - Bot Training**

For the training part, it consists of a training unit and deploying unit. So, before queuing another new unit into the unit building, it will first check whether there exist any deployable unit for each unit building that the bot owns during that round. If it exists, it will deploy the unit, if no, then it will continue into the training section. Inside the training section, it will first check the number of resources generated each round and it will also check the number of each specific type of unit that the bot owns. According to that, it will train a different type of unit.

**p21 Bot Algorithm - Construction**

For the construction part, it consists of an outpost which is going to be constructed outside the city and another building which is going to be constructed around the city range. For the building that can only be constructed inside the city range, at first, it will check the type of building that has been built around the city. Then it will construct the building according to the missing type and building priority. As for the Outpost, for every 2 owned cities, we will assign a unit to construct an outpost at the middle point of those 2 cities. The selected unit will construct the outpost when it arrives at the targeted location. To prevent any outpost to be constructed in an unwanted area, we use the middle point of the unit pathing system from the first city to the second city.

**p22 Bot Algorithm - Movement**

Move on to the movement part, there are 2 different type of movements, the first one for capturing neutral cities and the second one for capturing enemy cities. For capturing a neutral city, the bot will assign a single unit to move towards the nearest neutral city. As it arrived, the assigned unit will capture the targeted city.

**p23 Bot Algorithm - Movement (Cont’d)**

On the other hand, for capturing enemy city, We want to make sure if the bot’s unit composition is fully prepared before raiding enemy city. We want the bot to achieve a certain condition, such as, number of cities owned and number of units that are currently deployed. We assign a group of units to move toward the nearest enemy city. When the units arrive at the midpoint, they will build outpost which can help them to resupply their resources. When the enemy city is inside the unit range, the units will try eliminate every enemy unit around city before trying to capture the targeted city.

**p24 Bot Algorithm - Combat**

For the combat parts, we want the bot’s units to win as much as possible. Every time a unit found an enemy inside their attack range, we compare the number of enemy unit in range with the number of friendly units. If the number of enemy units fewer than the allied units, it will attack, if the number of enemy higher, it will retreat to any available friendly unit in a certain range. If it is not possible, it will attack instead.

Next, I’ll handover to Marcus to talk about our project User Interface.

**p25 Camera**

Thankyou Hubert.

In any game, the camera is an essential part as it determines what the player can see in the game world. And what the player can see would affect the gameplay and experiences. To ensure smooth gameplay, a camera system needs to be built and there are several criterias.

1. The camera should focus on the same object during rotation

2. The camera should be set that the player is always looking at the gameboard

3. The motion of camera should be smooth

**p26**

The concept of the built camera controller system is like creating a virtual pole from the target object and putting the camera on the other side, pointing to the target object. This approach allows us to focus on the target during rotation. It is perfect in strategy games where players will be focusing on the management in their territory.

We are using four variables to control the camera.

The first is the position of the target which determines the middle point of the camera view.

The length from camera to the target determines the zoom amount.

The angle of depression of the camera determines the degree of rotating up or down.

And horizontal rotation determines the degree of rotating left or right.

**p27**

To limit the movement area to avoid the player's camera not pointing to the

playmat, maximum and minimum value of these four variables are set. The limitations of these four variables are summarized in this list.

**p.28**

At this point, our camera controlling system has already attained the first two goals. And the one left is the smoothening of camera motion. A simple way of doing so is to perform linear interpolation every frame from current position and rotation to the target location and rotation. For our first attempt, we do linear interpolation on position of target, local position of camera and local rotation of camera but that creates a problem. If you look at the interpolated position on the picture, you may notice that the distance between the target and the interpolated position is shorter than normal, which creates an annoying zooming in and out effect.

**p.29**

To solve the problem, we have to be aware of all variables that players have control on when designing a smoothing algorithm. Therefore, we should do linear interpolation on the four variables and calculate the interpolated transform each frame to create a smooth motion.

**p.30 Tween**

For UI animations, we have adopted an approach called tweening. It is done by providing the keyframes and interpolating the frames in between. In this project, LeanTween library is used to create simple UI animations which cost less resources than Unity in-built Animators.

**p.31 PropEventTrigger**

As mentioned in the unit lifecycle, PropEventTrigger Class is implemented for providing a toggle on/off function to the props in game. The reason behind is that different behavior is expected even if the same event type is triggered in different scenarios. For example, players may click on an object if they want to examine the stats or they want the unit to perform an action. It is expensive to subscribe and unsubscribe to the event trigger. As the number of props scales with the map size which can be customized by players, reducing the resources consumed is significant in relieving the performance issue in the game.

**p.32 Menu Navigation**

For menu navigation, two different approaches are used in two scenes. In the main menu, the order of the menus are dependent on each other. So A stack is used to implement a web-like navigation where the back button is available.

**p.33**

In the battle scene, an event-driven system is used since the panels are independent of each other. When PointerOnClick events are triggered, the corresponding menu will be shown. For example, when a unit is clicked, the unit panel which provides stats of the unit will be shown.

Max will then talk about the technique used when creating models for the game.

**p.34 Models - Texture**

Thanks Marcus. I’ll now talk about the textures of the models. Textures define the appearance of models. Texture maps can be used to create more realistic looks of models and the environment. In our project, several types of texture maps were used. The first type we used is bump maps, which stores the “height” details of the texture image in grayscale. The darker the area the more it appears to be pushed into the surface. The second type we used is normal maps, which stores the surface normals in RGB format, and the three components correspond to the XYZ axis in 3D world. Both bump maps and normal maps do not actually modify the mesh of the 3D objects when applied, but instead they create illusions of bumpiness by telling how lightrays should interact with the surfaces of the models. The third type we used is roughness map, which describes the degree of light scattering on the surface in grayscale. The darker the area the less scattering of light rays it has. This is to emulate the roughness of real-life objects. The fourth type we used is cube map, which is a special type of texture map used for skybox. That is the background of our game. As the game world is actually a cube, cube maps store reflections on the environment in the form of an unwrapped cube, which consists of 6 squares.

**p.35 Models - Opt**

Next, I’ll talk about how we optimized the models. When we tried to add more details to the 3D models. The polygon counts increase. This triggers a problem: Higher polygon counts infers higher GPU loads, which in turn causes frame rates to drop and the game cannot run smoothly. This would definitely undermine the player experience. So in light of this, two solutions were used and they are decimating the models and applying Level of Details.

**p.36 Models - Opt - Decimate**

Decimating refers to the reduction of polygon counts, most preferably no obvious changes in appearance is resulted after the operation. Here is one of the rock models used for the stream terrain. Before decimating, the polygon count was over 3000. Yet after decimating, the polygon count was reduced to around 300, which is one-tenth of the original, while the appearance did not change significantly even on close-up view.

**p.37 Models - Opt - LOD**

Next, I’ll talk about the level of detail. The decimate solution mentioned previously may not suit every use case in our project. In some models like the grasses, the appearance on the close-up view changes a lot even with a small degree of decimation. In that case, level of detail is used. It only reduces polygon counts when the camera moves away to certain distances. In other words, it reduces polygon counts when the apparent size of the model drops below certain percentages of the original. Preferably, the appearance should not have much differences with the original one when viewed through the camera.

Here are the figures showing the grass models at different levels of detail, from left to right, they are level 0, 1, 2, and 3. The polygon counts of levels 1 ,2 and 3 are 70%, 40% and 10% of level 0. In close-up view, the models’ appearances change drastically. Yet when rendered in Unity and viewing from the camera, the changes are much less obvious.

Next, I’ll handover to Nathan to talk about our implementation of the Multiplayer feature.

**p.38 - Multiplayer (1) (slide with 1 picture)**

Any good strategy game needs to have a multiplayer aspect to enhance the player experience. It’s fun for people to play against their friends, or against other players in general. The multiplayer or networking portion focuses on allowing players to interact with each other, facilitated by our game.

**p.39 - Multiplayer (Our game’s multiplayer needs)**

The gameplay flow of Steel of Stalin is similar to the board game Diplomacy, simultaneous turns. Each player decides on their actions each turn, which are then executed simultaneously. These actions and interactions are then handled by the game, and the feedback is sent to the players. Thus, the interaction between players happens at the end of their respective turns.

As a result, our game must receive the player’s actions during the end of their turn, execute them, then return the information to display back to the player.

In addition, to maintain some realism, our game utilizes a common mechanic in many other strategy games called “Fog of War”. This mechanic hides information from players when they are far away, only revealing information that is close by. For example, a player cannot see the units that are far on the other side of the map, only spotting them when they get close.

**p.40 - Netcode for GameObjects:**

In this project, we used Netcode for GameObjects, Unity’s solution to multiplayer networking. It is a mid-level networking library built specifically for Unity.

Using Netcode for GameObjects, for multiple players to join the game, they must join a server. This server is run on one of the player’s computers. The player hosting is designated as a Host, and is essentially a Client attached to a Server. Other players are Clients connecting to the Server run on the host player’s computer.

The Client and the Server can communicate in 2 main ways: Remote Procedure Calls, or Named Messages. Named messages are longer and used for long, infrequent messages. Remote Procedure Calls are used for shorter, more frequent communication.

Let’s take a look at the flow of communication between the Server and the Clients.

**p.41 - Client-Server Communication:**

As mentioned earlier, communication between the Server and the Clients happens mostly at the end of the round, when the players submit their commands. So at the end of the round, Clients send their commands to the server. At this time, the server would receive the commands from all the Clients, and execute all of these commands. After executing all the commands, the data from the execution needs to be given back to the Clients to display. However, since the players are not supposed to know everything, and only what they can see due to the fog of war, the Server must filter the data to display to each Client. The filtered data is then sent to each respective client. After the Client receives the data, it displays it on the screen for the player to see.

Having displayed the data, the Client is ready for the next round. But since each Client is displaying different data, the time it takes for each Client to get ready for the next round may be different. To synchronize the starts of each round, each Client sends an “OK” to the server to signify that it is ready for the next round after it finishes displaying the data. The Server then will send a “Next Turn” command to all Clients after it has received an “OK” from all of them.

When the “Next Turn” command is sent to the Client, the next turn begins and the player can begin inputting commands for their next turn.

However, it’s not guaranteed that the Client and the Server will be able to communicate in this way every time. Whether it’s due to external circumstances or a game crash, we can’t wait forever for communication to occur, so we have implemented a timeout. At each step, if no response is received by the timeout, if it is a Client that cannot reach the server, the Client disconnects. If a Server cannot reach a single Client, it disconnects, and if a Server cannot reach any of it’s Clients, it shuts down.

**p.41 - Issues and Difficulties**

Although the flow is relatively straightforward, the implementation of the multiplayer was an extremely difficult task. There were a lot of issues with synchronization. For example, at the beginning of the game, when players join a lobby, all players must be sent a copy of the map. The amount of time it takes for players to receive the map is different, and the game start must be synced. Additionally, when initializing the network gameobjects such as: the network utilities, network manager, player object, the order of initializing them matters. This caused many different scenarios from clients desyncing from the server.

Another problem we faced was with the length constraint of the named message. Although the intention and purpose of the named message is for infrequent, long messages, we discovered there was a length limit to the named message. As such, very long messages such as the map data were not able to be sent via named message. To solve this problem, we created a custom RPC to send long messages. The RPC splits up the message into small chunks, sends them via RPC, and reassembles them in the correct order. We used this custom RPC to send long data regarding the game data such as BuildingData, UnitData, and the Map object.

**p.47**

Next up, we have our Live Demo!