Real Time Strategy Game

Graphics Unity Project 2022

Deadline: Sunday 14/06/2022

NORTHGARD, photo taken from pcgamesn.com

Contents

1 Introduction

The task is to create a single-player Real Time Strategy-like Game in **Unity** game engine.

RTS games vary in their core components, gameplay, theme and mechanics. In our case it is only required a subset of the mechanics and components that can be found on a fully developed and well designed RTS title.

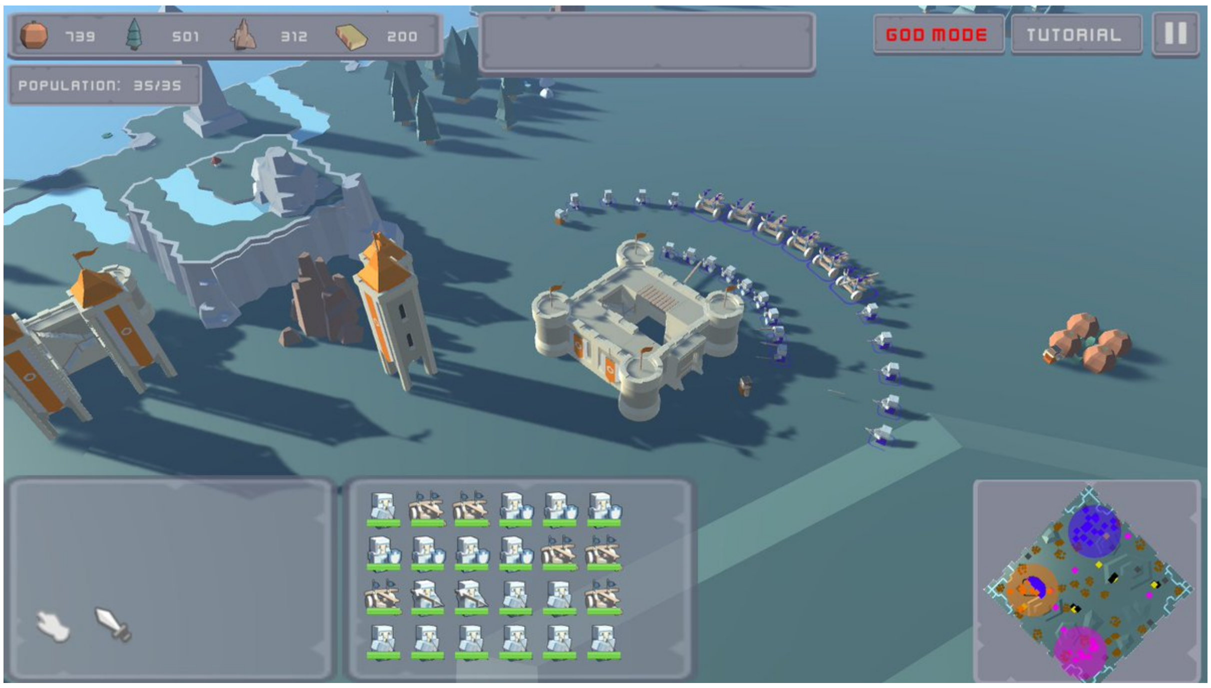
The required components and mechanics will be analyzed in the following sections. You are free to design and execute any of the requirements in your own way using your imagination and experience on other similar games. It is highly recommended to bring your own ideas on the table and take our notes as a guideline to implement this idea of yours.

A fully completed game (100% of marks) should consist of the following sections and Unity components:

1. **Graphics**: 3D scene design, 3D objects, textures, materials, lights.
2. **Physics**: colliders, rigidbodies, forces .
3. **Scripting**: game logic and mechanics, event handling.
4. **Audio**: sound effects, music.
5. **UI**: text display, buttons using Unity's UI components.

The more sophisticated the implementation of each of the above sections, the more marks your project will gain.

2 Game Overview & Core Components



RTS Engine Unity Asset: Photo taken from Oussama Bouanani

The concept of the Graphics Course's project for 2022, is associated with a very specific genre of games, the Real Time Strategy(RTS) Games.

What is an RTS Game

In a real-time strategy game, each participant positions structures and maneuvers multiple units under their indirect control to secure areas of the map and/or destroy their opponents' assets.

In a typical RTS game, it is possible to create additional units and structures, generally limited by a requirement to expend accumulated resources. These resources are in turn garnered by controlling special points on the map and/or possessing certain types of units and structures devoted to this purpose.

More specifically, the typical game in the RTS genre features resource-gathering, base-building, in-game technological development, and indirect control of units. On top of these core gameplay features, a lot of functionality can be built. From "smart" AI opponents that assault your bases and buildings, to a more peaceful agriculture society that its only goal is to progress economically, technologically and culturally. The possibilities are endless and sky is the limit in the RTS genre.

If you are not familiar with the genre it is highly recommended to watch 1-2 videos of RTS gameplay to get the general idea.

- Video Gameplay - Warcraft 3: Elves vs Trolls.
- Video Gameplay - Age of Mythology.

Core Components of the Project

In your project it is required by you to create the basic functionality of a SIngle-Player RTS game. The game should follow the basic guidelines of an RTS game and the gameplay to be relevant to the genre.

The main features that should be developed are:

2.1 Units & Commands

Units: The player should be able to spawn/create units during the procedure of the game. Units can be of different types and have various interactions. The number of your units should be finite and have some boundaries based on other aspects of your game. Unit is considered every object which can receive and execute a list of commands.

Commands: The player should be able to give commands to a selected set of units. Commands is every action that the unit can execute and have a specific result, for example:

- **Move to Destination**: Player Commands the unit/s to move to a specific destination.
- **Interact with other Units**: Player commands the unit/s to interact with the pointed target unit (e.g attack an enemy / heal an ally unit etc).
- **Interact with Environment**: Player commands the unit/s to interact with a doodad (e.g. tree / rock).
- **Interact with Buildings**: Player commands the unit/s to interact with a Building (it can be ally or enemy building).
- **Build Building**: Player commands the unit/s to build a building.
- **Transfer Resources**: Player commands the unit/s to transfer resources from one point to another (e.g. from a building to another).

These are the main commands you can choose from, adjust them to your game needs and gameplay.

2.2 AI Components

AI Navigation: The movement of your units should have some path-finding logic, to move from one point to another in a ”smart” way. For example a unity cannot got from point A to point B by passing through a wall or building. You can achieve this functionality by using the Unity’s NavMesh System.

Command Execution: For any given command there should be a logic that the unit is following in order to execute the command. For example: if the user commands the unit to interact with an object the unit firstly must move close to the target, and then if it’s an intractable object to interact with it. After the interaction has ended the unit awaits for new commands in an idle state (this is just an example, this means you can implement this command with different logic if it makes more sense in your game).

2.3 Resources

Resources: Resources have major impact in your game, as it must be the main currency in your game. You can have as many types of resources as you want. If you choose to add any type of resource make sure that there is at least one way to produce/gather it and one way to consume/lose it while playing the game.

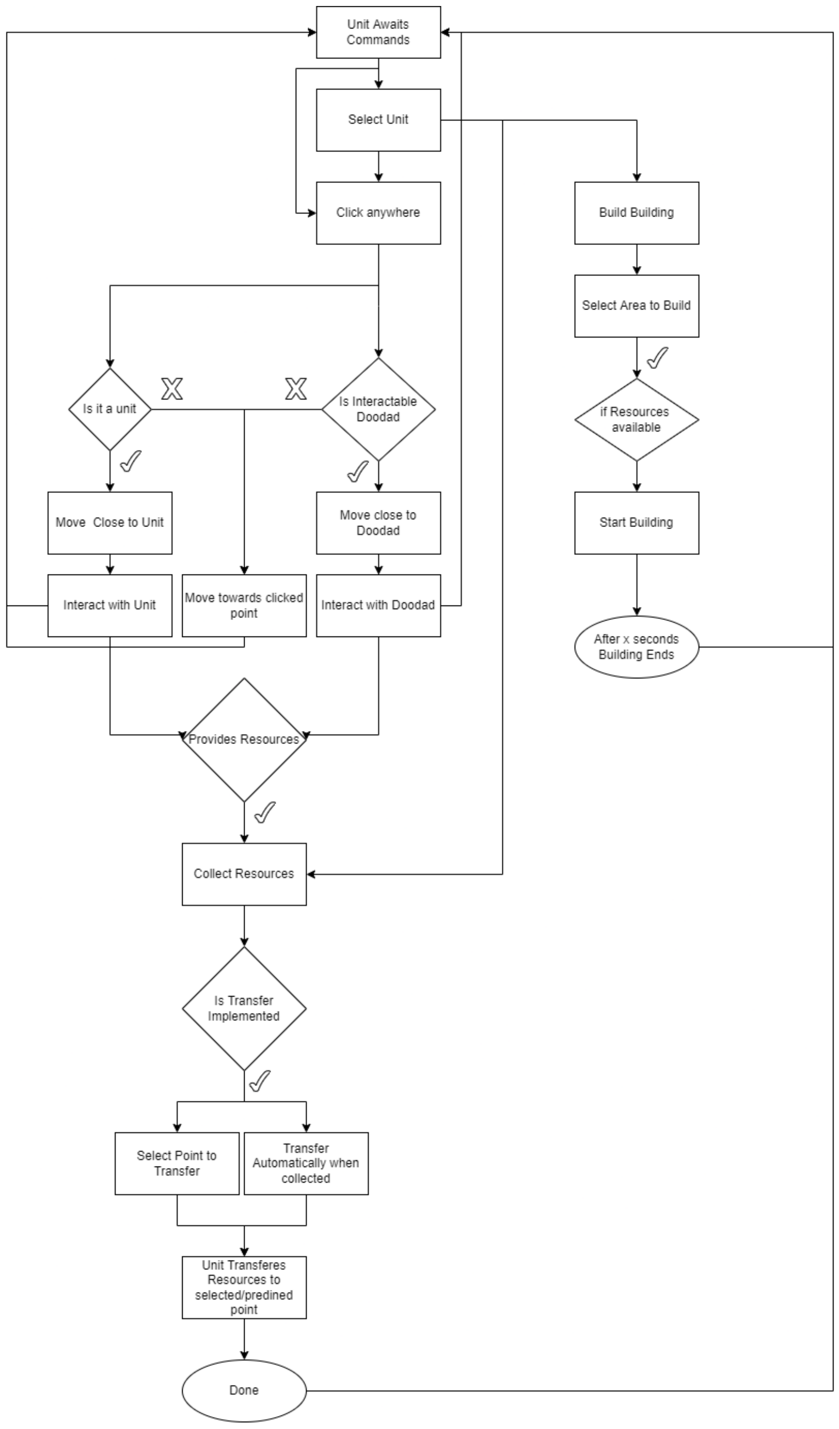
Examples of Resources and use cases:

- **Wood**: Gathered from trees. Spent to build buildings [2.4].
- **Population**: The current value is increased by 1 for every unit the player controls [2.1].

- **Gold**: Gathered from gold mines. Spent to buy units [2.1].
- **Bread**: Bakery building [2.4] produce one bread every 5 seconds. Every 30 seconds lose one bread for every unit under the player's control. If there is not enough bread for all units, excess units are lost (Population decreases).

These are only some example resources, you can create your own based on the style of your game and your goal.

Units Command Execution Diagram: An example of the sequence in which a command may be executed.

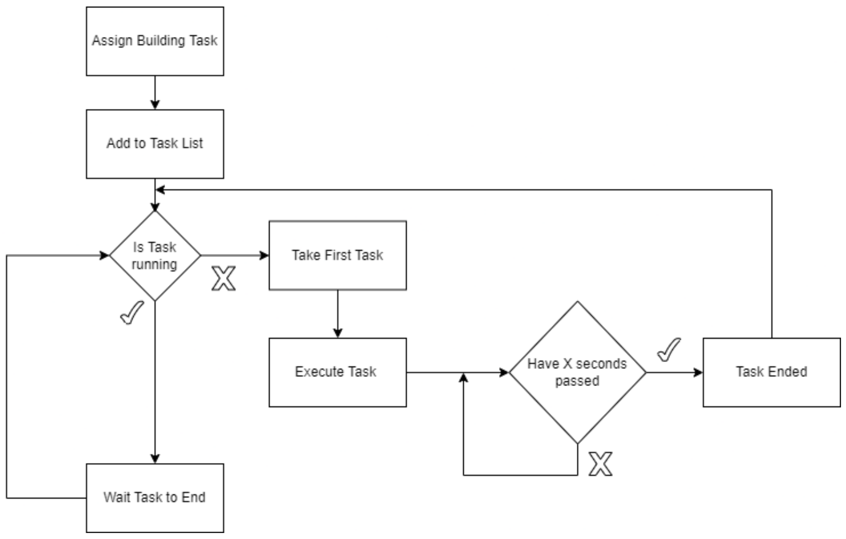


2.4 Buildings & Environment

Buildings: In your game the player should be able to build and interact with different kinds of buildings. Buildings are objects in your game that have a specific purpose and don't act like units. The different types of buildings that will be available in your game should have different functionalities. For example a building might be responsible for the production of units. Another one might be responsible to upgrade your unit numbers or the consumption of resources by other buildings etc.

Buildings Functionalities: The functionalities of your buildings should be relative to your game's goal and design. Player must be able to assign task in the building (e.g. produce 8 units). The assignment/task should cost some of your selected resources(one/more) or any other transaction that you might think (e.g. sacrifice units to produce a bigger/stronger one). If the player gives two or more assignments/tasks, then they must be executed in chronological order, the one after the other, and not simultaneously (in the example: the 8 units should be produced one by one).

Building Tasks Diagram: An example of the sequence in which a building task may be executed.



Structures: Structures are the buildings that have no extra functionality other than be an obstacle or provide visual update to your current game. For example the player might be able to build walls around a base, or fence around a farm (walls and fence act as static structures here).

Doodads: Doodads are the objects which are placed during the creation of a level and might act as intractable objects for your units (e.g. trees, rocks, mines, plants, animals etc.).

2.5 Level Design

Levels: The way you are gonna design your game levels may affect the way that your game is being played. Design your level like a map, where the player should experience your game. Make sure your levels have the same aesthetic/feeling with the rest of the game. For example if you design a sci-fi game, it makes no sense your level to look like a medieval battlefield. You might wanna introduce some verticality in your game with higher and lower grounds. Units on high grounds might have advantage in battles for example.

A very nice approach is to let your levels have some randomness during the creation. In this way the player never plays the same map again and again. It is highly recommended to implement a random generated level design. The more sophisticated the implementation the higher the reward and the more unique levels you can produce. Some issues that may occur that are not desired:

- Spawning objects on top of each other.
- Not spawning enough objects required to compete the game.

Your levels should be playable, and there should be enough objects for the player to interact with.

2.6 Core Mechanics

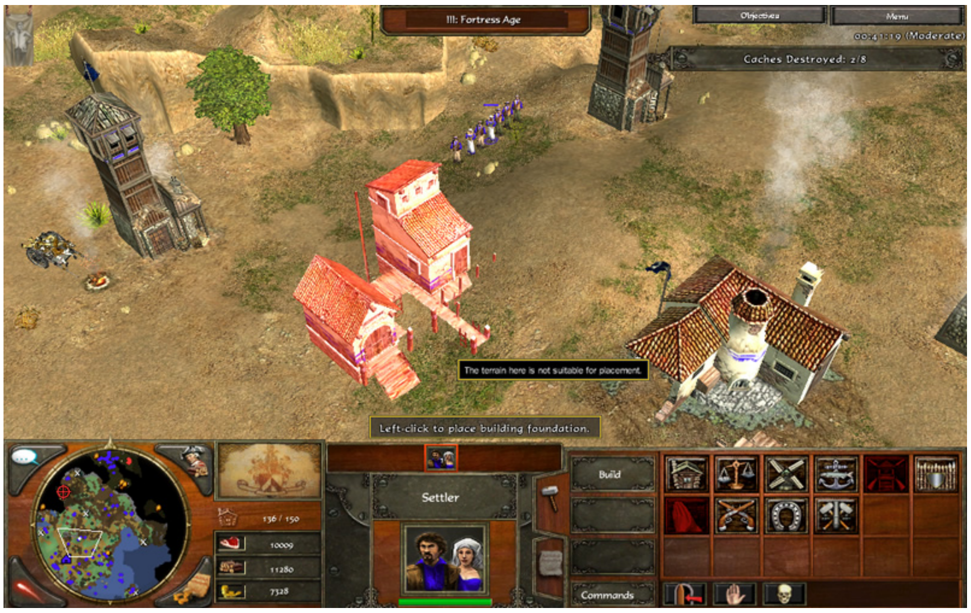
Camera Control: You should have at least one controller for camera handling. The user/player should be able to view every aspect of the game using the camera. Many RTS games use an isometric point-of-view, but you are free to implement any point-of-view that you think fits into your game style. It is highly recommended to implement more than one point-of-views or let the user change the cameras options at will. This will help during the play-test phase.

Multi-Unit Selection & Command: The player should be able to select any number of the available units[2.1] and give them a command. It is recommended to implement the functionality using primitive shapes at first to make sure you have implemented it correctly at any circumstances.

2.7 User Interface

UI: RTS games in general, are heavy UI focused. A lot of interactions given by the player comes from different UI elements. You should implement the basic UI functionality required for your game to be played smoothly. Don't take anything for granted and explain everything in your game using UIs or by guiding your players in a smart way.

Show at every state of the game the progression to the player, and communicate with him every info that affects his/her experience throughout the entire procedure (e.g. show resources available, buildings that are being built etc).



Age of Empires 3 photo taken from Edd Grimes Media Work

3 Requirements

Game Design Document: A game design document outlines the core of your game in a few bullet points. Your game design document will be the main part of your report. Examples on how to design a game design document are in section 5.

3.1 Minimum Requirements (50%)

In order to pass the graphics course (5/10) you have to implement at least the minimum requirements. You are not allowed to use game-ready plugins/packages that implement the functionality of components that are listed below. For example you can not use a package that does the whole navigation system by itself for you, but you can use game-ready 3D Model assets and animations. You can use UI assets for UI/buttons but not a whole game-ready UI system package.

1. **Core Mechanics (5%)**:
   - Camera control: A script that moves the camera. As a minimum the camera should move along two axis with an extra condition to not move too far away from the playable area.
   - Multi-unit selection: The player should be able to select and command any number of units at once.
2. **Units, Commands & AI (10%)**:
   - Two or more units with different functionality. A Unit is considered any object which can receive various commands, which it executes based on an underlying AI system.
   - AI navigation: Units should have some pathfinding logic to move towards their destination.
   - At least three (3) commands and command execution logic for the given command. One of them should be movement command (See Commands Section 2.1).
3. **Buildings (10%)**:
   - At least two (2) different kind of buildings with different functionalities. At least one building should be able to build new units.
   - Overtime task executions. Creating/upgrading buildings and task executions on buildings must be completed over time (a few seconds), not instantly.
   - At least one (1) doodad capable of interacting with units (e.g. trees to gather wood or enemies).
4. **Resources (2.5%)**:
   - At least two (2) resource types. The player should be able to produce/gather and consume/lose resources while playing the game.
5. **Level Design (7.5%)**:
   - Random generated objects on map. As a minimum, the player and miscellaneous objects like enemies and resources/doodads should be placed randomly inside a map.
   - **ALTERNATIVELY:** At least three (3) pre-designed and static maps. Also, allow the player to select any map from the main menu at all times.
6. **UIs (10%)**:

- Main menu: When the game first opens, it should start with a main menu with at least one button to start the game and another to quit the game.
- Resource counters: All resources with numerical values should be visible at all times.
- Building UIs: Each building should have an appropriate UI for interacting with it or viewing its information when selected.

7. **Credits & Audio (5%)**:
   - The credits should include a list of all external resources used including all 2D and 3D objects you did not make yourselves with a link to their source, a list of all the tutorials you saw on the internet and another list of all the places you took code from.
   - Audio: At least one ambient background music playing automatically and looped during the game.

- In every Component the visuals and graphics take a part of the marks. At the bare minimum all your objects should have a material with an attached basic texture. If a project share the same functionality with another one, but is visually better it will get more points from each individual component.
- Use colliders and rigidbodies (if needed) to apply physics to your objects so that they behave naturally and as realistic as possible.

3.2 Extra Requirements(50%) + Bonus(10%)

You must complete the implementation of minimum requirements first, to start the implementation of the extra requirements.

1. **Core Mechanics (5%)**:
   - More than one point of view and zoom in/out.
   - Functionality additionally to multi-selection feature (e.g. player can create sub teams to control units or select all type-x of units at once).
2. **Units,Commands & AI (12.5%)**:
   - Additional 2 type of units with different functionality.
   - Additional 2 commands with command logic. (See Commands Section 2.1).
   - At least one(1) type of AI NPCs with automated behaviour.
3. **Resources (2.5%)**:
   - Additional two(2) more type of resources.
4. **Buildings (7.5%)**:
   - Additional two(2) types of buildings with different functionality.
5. **Level Design (10%)**:
   - Procedural/Fully Random Generated Maps.
   - Fog of War effect (see link to get a better idea of what it is)
6. **UIs (10%)**:

- Minimap on the Screen.
- Pause menu.

7. **Audio (2.5%)**:
   - Additional Music Tracks.
   - Sound Effects: sound triggers when an event happens.
8. **Any Extra Functionality/Effort (BONUS 10%)**:
   - Bonus can also be given in cases where the implementation of a feature is extremely well-developed or very creative/unique.
   - Self made 3D/2D assets, Music, Videos count as extra effort and give extra bonus. You must clearly state which assets were custom made and the software you made them with in your credits.

4 Useful tips

- It is highly recommended to give your executable version of your game to be tested by others without your guidance. Try to evaluate your game utilizing the thinking-aloud method. In a thinking aloud test, you ask test participants to use the system while continuously thinking out loud — that is, simply verbalizing their thoughts as they move through the user interface.

  Don't trust your instincts on the quality of your UIs implementation. User experience is crucial during the testing of your game. Put effort in the creation of your UI/UX and make it as intuitive as possible.

- During the exam, you will be graded based on the playable version of your game. Anything you implement that cannot be shown while playing the game will not be graded. As such you should take into consideration:

  – Someone playing the game should be able to see everything you want to show in a maximum of 10 minutes. You will be able to give us advice while playing your game in order to achieve this.

- Excluding extreme circumstances we will not look at your inspector and we will not view your project through the Unity inspector so make sure your build is fully playable! Some common issues that may appear include the following:

  – UI elements may not scale correctly with your screen resolution and be out of the playable field. Make sure your game plays correctly at least for the recommended resolution of Full HD (1920x1080) or smaller. If you have a specific resolution you tested your game at and it works correctly other than Full HD specify it on your report.

  – Sometimes things that work in the Editor do not work the same way after you build the game. Always build and test your game during development! The most common issues involve the initialization of objects while switching between scenes e.g. when transitioning from the main menu to the game or when restarting the game and reloading the same scene.

- **Using a piece of code which you cannot explain will not give you any grade for that specific section!!!** It is recommended you watch online tutorials and even exchange ideas with each other but plagiarism is strictly forbidden. Additionally, custom and creative solutions that work will receive bonus grades. This means that if you get some piece of code from the internet and you cannot explain it, if you put additional effort in another field you can compensate for the grades you lost and still get a good mark.

5 Game design

The core game design dictates what the player expects to see in the game in three sentences. It is a very broad guideline that dictates the core design principles of the development process and helps guide us when coming up with new ideas or when searching for assets.

The core design includes three topics: Player (Who is the player?), Setting (In what location and time period does the game take place? What is the main visual theme or technological background?) and Goal (what does the player seek to achieve and how?).

For strategy games, another thing included during game design is a unit/building look up table. There, each unit's capabilities are explained in a plaintext format (see examples below).

5.1 Example game 1 – Caveman survival

- **Player**: The player controls a caveman tribe out in the Savannah.
- **Setting**: Prehistoric age. African wasteland.
- **Goal**: The player must build a colony and fight off predators. The player wins if they kill 20 wolves.

Resource: wood	Gathered by interacting with trees. Used to build buildings.
Resource: food	Gathered by interacting with berry bushes and animal carcasses. Used to build new units
Resource: population	Maximum population is increased for every hut built. One population is required for every unit built. Maximum of 50 population allowed.
Unit: Caveman craftsman	The caveman craftsman can gather wood and food from bushes. Craftsmen can also build huts and barracks.
Unit: Caveman hunter	The caveman hunter can attack wild animals and gather food from animal carcasses.
Building: Hut	Requires wood to build. Gives max population.
Enemy Unit (AI): Wolf	Moves towards and attacks nearest enemy if they come within a certain radius around it. Leaves behind an animal carcass when killed.
Doodad : Tree	A craftsman can gather wood by interacting with a tree. The tree is then destroyed.
Doodad : Berry bush	A craftsman can gather food by interacting with a berry bush. The bush is then destroyed.
Doodad : Animal carcass	A hunter can gather food by interacting with a carcass. The carcass is then destroyed.

5.2 Example game 2 – Gold rush

- **Player**: The player is a rich investor leading a mining team to gather gold.
- **Setting**: California Gold rush era (1848-1855), California.
- **Goal**: Gather minerals and identify them to obtain golden nuggets. Gather as many nuggets as possible in a short amount of time.

Resource: Unidentified ore	Gathered by prospectors by interacting with rocks. Placed on ore deposit piles by prospectors. Transferred from ore deposits to identification stations by pack mules. Consumed to build buildings and buying units.
Resource: Gold nugget	Has a chance to be obtained by an inspector when identifying rocks. Transferred from identification stations to vaults. Gold nuggets in vaults are used for the final score of the player.
Building: Recruitment office	One recruitment office is spawned at the start of the game. Consume ores to recruit new prospectors or pack mules through this building.
Building: Ore deposit pile	Contains unidentified ores. Can hold a finite amount of minerals.Prospectors deposit unidentified minerals in ore deposits. Pack mules pick up ores and transfer them to identification stations When consuming unidentified ores to recruit units or building, the ores are removed from any ore deposit pile where they are available.
Building: Identification station	Transforms unidentified ores into gold nuggets. Every 5 seconds consumes 1 unidentified ore and has a 30percent chance to produce one gold nugget. Can hold a finite number of unidentified ores and gold nuggets. Pack mules deposit unidentified ores onto identification stations. Pack mules pick up gold nuggets from identification stations
Building: Vault	Contains golden nuggets. Only golden nuggets deposited into vaults are taken into account for the final score of the game. Pack mules deposit gold nuggets into vaults.
Unit: Prospector	Can hold up to 5 unidentified ores. Gathers ores from ore veins. Deposits ores into ore deposit piles. Builds deposit piles, identification stations and vaults using unidentified ores.
Unit: Pack mule	Can hold up to 5 unidentified ores OR gold nuggets. Picks up unidentified ores from ore deposit piles. Deposits unidentified ores and picks up gold nuggets from identification stations. Deposits gold nuggets to vaults.
Doodad: Rock pile	Interactable by prospectors. Gives 1-3 unidentified ores and destroys self on interact.

6 Basic Instructions

- Use your imagination and creativity to create a good-looking scene and scenario.
- You must complete the implementation of minimum requirements first, in order to start the implementation of the extra requirements.
- You can use Unity's **3D models** (cubes, spheres, ..) to create your scene by transforming them to give them the shape you wish.
- You can create your own 3d models in separate 3d-modeling software (such as 3D Studio Max, Blender, etc) and import them in your Unity project. However, if you don't have previous experience with 3D modelling, this option is not recommended. For Basic shapes and level designs you can use the ProBuilder Unity package, to create simple shapes withing unity editor.
- You can import third-party 3d models found on the Internet (Sketchfab) or Unity's Asset Store to help you build your scene. Remember: If you download and import third party 3D models you should include them in Credits.
- You can use the Built-in "Collaboration" tool in Unity to use for version control in your project (similar to Github).
- You must add appropriate **materials** to object surfaces to give them a realistic feel and look. Use Unity's Material Editor accordingly.
- It is highly recommended to use the Unity Documentation as it provides both explanations on all of Unity's tools as well as examples on how to use them. When you open the Documentation remember to select your Unity Version on the top left.

7 Physics

7.1 Collision between objects

Objects must have **Collider** components attached to them, otherwise they will pass through each other. It is important that the collider matches the 3D shape of an object as best as possible. For example, a box collider is ok for a box-shaped object, such as a wall or a smooth floor. More complicated 3D models require different types of colliders. For example, using a box or a sphere collider for a pin is wrong, as it will result in un-realistic behaviour.

Documentation - Collision between objects

7.2 Rigidbody

Rigidbodies enable your GameObjects to act under the control of physics. The **Rigidbody** can receive forces and torque to make your objects move in a realistic way. Any GameObject must contain a Rigidbody to be influenced by gravity, act under added forces via scripting, or interact with other objects through the NVIDIA PhysX physics engine.

A rigidBody is defined by many parameters, such as its mass, drag, angular drag and more.

Documentation - Rigidbody

7.3 Physic material

The Physic Material is used to adjust **friction** and bouncing effects of colliding objects.

To create a Physic Material select *Assets, Create* , *Physic Material* from the menu bar. Then drag the Physic Material from the Project View onto a Collider in the scene.

Documentation - Physic Material

7.4 Realistic physics

Game objects affected by physics (rigidbodies) have parameters, such as mass, drag and angular drag. Set their values accordingly by experimenting, so that they behave realistically.

8 Scripting: Game logic and functionality

Scripts control the entire functionality of your game, such as handling user input, triggering events upon specific actions, keeping score, moving objects and more.

Your scripts will have to:

- **Initialize** variables, load game objects and/or other scripts that need to be accessed.
- **Keep track of the game state** and act accordingly. Check for key presses, position changes, collisions and more...
- **Reset game objects' state** (such as position, rotation, gravity, ...) **when required**.
- Update the UI elements when necessary. For example, have a UI text that shows the current gold the player has.

Important things to remember:

Code in Unity is written in C#. Every new script **derives** from the *MonoBehaviour* base class by default. A script deriving from *MonoBehaviour* enables the call of the Start(), Awake(), Update(), FixedUpdate(), and OnGUI() functions and has the following properties:

- A script will never be executed, unless it is attached to at least one game object in Unity. Furthermore, a single script can be attached to one or more game objects. In this case, **be careful**, as the same script will run as many times as the number of game objects it has been attached to.
- One GameObject can have multiple scripts attached, too. In this case, **be careful** of possible race conditions. (a race condition can occur when, for example, 2 different scripts try to move the same object at the same time).
- Game objects (or other scripts) need to be initialised in the Start() function of the script before use, unless the script is attached to them, in which case they can be accessed directly.

  The links contain useful tutorials on the communication between scripts and Game Objects in Unity:
  $Link_1$ $Link_2$

9 UI and Audio

9.1 User Interface (UI)

Unity's UI system provides a variety of tools to render elements, such as text or images, on the screen or at a specific position in the 3D space. The UI system can be used for many different purposes.

Documentation - User Interface

It is totally up to you to choose the design, position and functionality of the UI system - as long as it meets the minimum requirements.

9.2 Audio

You can use Unity's audio components to play sounds in a loop(ex. background music) or if a condition is true.

Documentation - Audio

10 Project Guidelines

- Your code must be organized in different scripts, depending on the functionality. **Each script must have a distinct role.**

- Use script names that make sense and attach them to related game objects. For example, create a "UIManager" script to handle the game UI, a "GameManager" script to hold variables and objects that keep track of the game's state.
- Your code must be written in C# programming language ONLY.
- Organize your game objects in a hierarchy that makes sense and is easy to understand. For example, if you have game objects such as "Cube(1)", "Cube(2)", "Cube(3)", ... , make them children of a single "CUBES" game object.
- Organize your asset files (textures, 3d models, materials, scripts, sounds, ...) in separate folders. Obviously, the name of the folder should be representative of the content.
- You may use the Internet (YouTube, blogs, ..) to your help for tutorials and code examples, but copy-pasting entire code or functionality is obvious and will NOT be accepted.

and remember while doing this project to **have fun !**