COMP2160 Game Development Task 2

Update Oct 18:

- Fixed Github classroom link (twice!)

Update Oct 12:

- Edited requirements 1(g) and 1(h) to simplify.
- Added Prefab-based scene construction to marking rubric.
- Specified Unity version
- “main” branch in GitHub will be marked

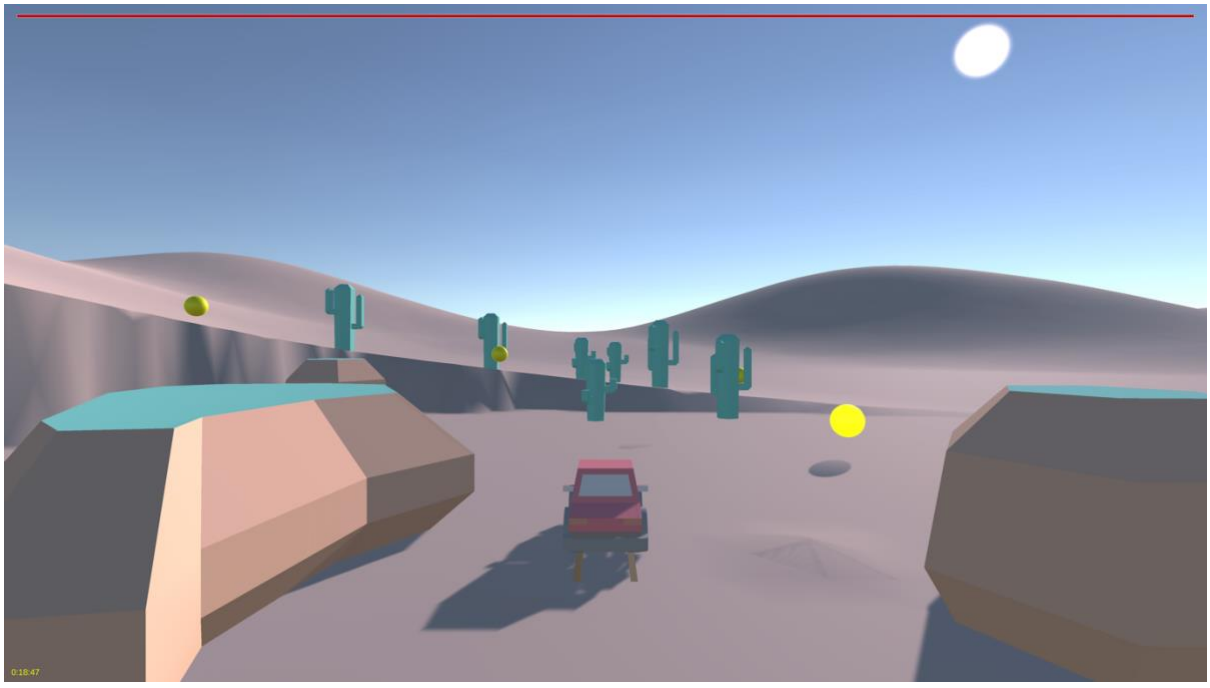
Topics covered:

- Physics
- Camera control
- Game Architecture
- Version Control
- Quality Assurance

Requirements

This assignment is to be done in pairs to given you experience with collaborative development. You will work in **Unity 2020.3.12**. Your task is to implement a copy of this 3D off-road driving simulator:

https://wordsonplay.itch.io/comp2160-assignment-2



Your game should implement the following features, described in detail below:

1. Physics-based car movement (10%)
2. 3D Terrain & Obstacles (5%)

3. Checkpoints (5%)
4. Player health (5%)
5. Camera control (5%)
6. User interface (5%)
7. Analytics (5%)

The marks for completing each feature are as shown. Completing each of these features comprises 40% of your mark (see Marking section below). Details of the feature requirements are below. Where the spec refers to a "tuneable value" this value should be made into a tuneable parameter in the Unity Inspector, with an appropriate value.

Framework

There is no framework provided for this project. You are expected to create the project from scratch. However two asset packs of 3D models are provided on the iLearn site for use in your project. These packs are provided by Kenney.NL under a Creative Commons 1.0 Universal license, so are free to use in your game without restriction. You are expected to use these assets to represent the car and obstacles in the terrain.

Features

1. Physics-based car movement (10%)

The game should simulate the movement of a car using physics.

a) The car should be represented using one of the vehicle models provided on iLearn, with appropriate textures and collider.
b) Forward and backwards movement is controlled with the W and S keys:
   - **W** – accelerate forwards with a fixed force
   - **S** – brake / reverse with a fixed force
c) Steering is controlled with the A and D keys:
   - **A / D** – turn left/right with a fixed turning circle radius
d) Turning while the car is in reverse should behave realistically, i.e. if 'right' and 'reverse' are held simultaneously, the car should rotate anticlockwise (see reference implementation).
e) The car should be affected by gravity, a constant force in the world's downward direction.
f) The car should be affected by a linear drag force opposite to the direction of movement of the car, proportional to its velocity.
g) The car should be affected by friction when it is in contact with the ground.
h) The car cannot be controlled when it is in the air (i.e. more than a threshold distance above the ground).

2. 3D Terrain & Obstacles (5%)

a) The game should use standard Unity 3D terrain.
b) Static obstacles from the "nature pack" of 3D models (e.g. trees and rocks) should be scattered throughout the world.
c) The player can collide with these objects and will bounce off them.

3. Checkpoints (5%)

The player is required to travel to a series of checkpoints in a specific order.

a) Each checkpoint marked by an indicator that lights up when the checkpoint is active and dims when it is inactive.
b) At any point in time there is a single active checkpoint.

c) When the player passes within a tuneable radius of the checkpoint, it becomes inactive, and the next checkpoint in the sequence becomes active.
d) The game is over when the player reaches the last checkpoint.

4. Player health (5%)

The player has a health stat which works as follows:

a) The player starts the race with 100% health.
b) Colliding with either an obstacle or with the terrain reduces health, proportional to the force of the collision. Collisions below a tuneable threshold force cause no damage.
c) A tuneable amount of health is restored at each checkpoint.
d) If the health drops below a tuneable threshold, the car begins to emit smoke.
e) If the health reaches 0%, the car explodes, and the game is over.

5. Camera control (5%)

The camera follows the car with the following properties:

a) Horizontally, the car is always positioned in the middle of the screen (except when turning).
b) Vertically, the car's position depends on its velocity. When the car is stationary, it a close to the bottom of the screen. When the car accelerates forward, it moves further away from the camera.
c) When the car turns, the camera should be positioned inside the turn. i.e when the car is turning right, the camera should be on right-hand side of the car.
d) The camera maintains a vertical orientation (in world coordinates) regardless of how the car is rotated (e.g. if the car is driving on a slope)
e) The camera should move smoothly between these positions as the car accelerates and turns.

Storyboards are provided on iLearn to illustrate how this should work.

6. User interface (5%)

The game should run on any screen with a 16:9 aspect ratio.

During the game, the user interface should include:

a) A health bar showing the car's health as a proportion of the maximum health.
b) A timer showing the time since the start of the race, in the format "minutes:seconds:hundredths". E.g. "5:04:25" means 5 minutes and 4.25 seconds.

Upon game over a panel should appear showing:

c) Either "You win!" or "You died!" depending on whether the player completed the race or exploded.
d) A list of checkpoints completed and the time at which the player reached each. Incomplete checkpoints should be labelled "Incomplete"
e) A button allowing the player to start the race again.

7. Analytics (5%)

Your game should use the Unity Analytics server to record the following events and analytics:

a) The start of the game (using a standard game_start event)
b) The end of the game (using a standard game_over event)
c) When the player reaches a checkpoint:
    - The time since the beginning of the race

- The player's health (before the checkpoint health boost)

d) When the player dies:
   - The time since the beginning of the race.
   - The player's position in the world.
   - The name of the object the player collided with.

Documentation

Your project should include a Documentation folder containing the following documents:

- Entity Relationship Diagram
- Quality Assurance Plan
- Bug History
- Task Allocation

Entity Relationship Diagram

Your submission should include a PDF file containing an Entity Relationship Diagram, following the examples provided in lectures and pracs. The diagram should document the structure of your game code, answering the questions:

- What are the different objects (entities) in your game code?
- What parts of the game state are they responsible for?
- What public parameters are available on each object?
- How do they update their state?
- What state needs to be visible to other objects (read only)?
- What events do they respond to?
- What events do they send to other objects?

Quality Assurance Plan

Your submission should include a QA Plan spreadsheet listing a thorough set of tests for all the gameplay features above. Each test should detail:

a) A test ID
b) The ID of the requirement it is related to.
c) The scene that should be used for testing
d) A detailed list of instructions for the tester to follow in the game.
e) A description of the expected outcome of the actions.

For features that are time-consuming to test (e.g. completing the game) or difficult to achieve you should create specific test scenes to make them simpler, rather than do all testing in the main scene.

Bug history

You should test your game at multiple stages of development as new features are implemented. Your submission should include a Bug History spreadsheet listing the bugs you encountered. Each entry should contain:

a) The bug ID
b) The date of the test
c) The **severity** of the bug:
   Blocker (0) – Blocks testing and there is no workaround

Crash Bug (1) – Crashes the game but there may be a workaround
Critical Bug (2) – Major functionality problem
Minor Bug (3) – Error that is noticeable but not critical
Feature Request (4) – Desirable change but not currently a defined feature

d) The **component** of the game affected.
e) A short **summary** of the bug
f) The ID of the **test** that generated the bug
g) A list of steps to take to **reproduce** the bug (if possible)
h) What you **expect** the game to do
i) What **actually happens**.
j) A filename for a **screenshot** of the bug (if possible).

Any bugs found through exploratory QA should be added as tasks to your QA plan so they can be re-checked in future.

Team development and version control

This is a pair project but will be marked individually based on:

- Task allocation
- Contributions to the project git repository
- Peer assessment

One week before the final deadline you will submit a document, using the template provided on iLearn, indicating which tasks are to be completed by each team member. Tasks can include:

- Designing the code architecture
- Developing code for specific features
- Conducting QA
- Writing documentation
- Other important development or production tasks

A second copy of this document should be included in your final submission indicating which tasks were completed by each team member, including any additional tasks that were added after the original submission.

You are expected to use the **branching workflow** described in lectures for your repository. Your repo should include a *main* branch for stable feature releases (after testing is completed) and separate *develop* branches for each team member to contain features under development.

You are expected to use the **prefab-based scene construction** described in lectures to break the scene into separate components to avoid merge conflicts.

At the end of the project, you will submit individual **peer assessment reports**, using the template provided on iLearn, to assess both your own contribution and that of your teammate. You need to provide a grade (following the rubric given in the template) and a justification for the grade. This grade will be kept private from your teammate but will be used as evidence to adjust the final individual grade weighting.

Submission

Your will need to submit your planned task allocation **one week before the final deadline**, using the submission link on iLearn.

Your Unity project will be submitted via GitHub classroom, using the following link:

https://classroom.github.com/a/OGDrD4Dd

Note that there is no framework provided for this project. You are expected to create the project from scratch.

Make sure your GitHub account is correctly associated with your student number in GitHub Classroom. Edit the **README.md** file in your GitHub repository to include your names and student numbers, so we know it is your code.

A snapshot of your git repository will be taken at the assignment deadline. This is the code that will be marked. Later commits will not be marked except in a case of special consideration. Remember to commit and push your code early and often. No sympathy will be shown if your first commit is minutes before the deadline and something goes wrong in the process. Good version control habits are an important part of game development.

Note: only the code in the **main branch** of the repository will be marked. However, we will look at the commits made in the other branches to verify that the branching workflow has been followed.

The required documentation should be included in a **Documentation** folder in your project:

a) Task allocation document (revised as necessary to include final allocation of tasks)
b) Code architecture document
c) QA plan spreadsheet
d) Bug report spreadsheet

Your peer assessment should be submitted individually through the iLearn submission link.

Marking

Your mark consists of the following parts:

Completeness: 40%
Architecture documentation: 15%
Quality Assurance: 15%
Code quality: 15%
Task allocation and Version control: 15%

Marks will be individually moderated based on the criteria described above.

The rubric for documentation and code quality is:

Grade	Architecture documentation	Quality assurance	Code Quality	Task allocation & Version control
A+ (100%)	Excellent work. Thorough, detailed and clear ER diagram, accurately describing all the relevant objects in the game and the relationships between them, and answering all the questions above.	A thorough testing plan covering all the required features, with detailed instructions and expectations. Includes tests for unusual corner cases. Specific test scenes provided where appropriate. A history of bug testing throughout the development of the project. Clear descriptions of bugs with detailed instructions for reproduction.	Full compliance with the C# Style Guide. Code is easily readable. Well-designed code architecture. Appropriate encapsulation of private state. Designer-friendly code with all appropriate public parameters available in Inspector.	Task allocation shows clear allocation of all necessary tasks. Tasks descriptions clear and professional. Workload is carefully balanced. Feature-branching version control workflow correctly implemented with clear distinction between develop and main branches. Work committed frequently in meaningful chunks. Professional commit messages that meaningfully document changes made. Correct use of .gitignore and Git LFS with Unity. Prefab-based scene construction used to avoid merge conflicts in scene edits.
B (80%)	Very good work. A detailed ER diagram that accurately represents all the relevant objects in your game and the relationships between them. Some incompleteness in terms of detail.	A testing plan covering all the required features, with detailed instructions and expectations. A history of bug testing throughout the development of the project. Clear descriptions of bugs with detailed instructions for reproduction.	Full compliance with the C# Style Guide. Code is readable with no significant code-smell. Code architecture is adequate but could be improved.	Task allocation shows clear allocation of major tasks. Tasks clearly described. Workload is balanced. Feature-branching version control workflow implemented correctly. Work committed regularly in meaningful chunks. Commit messages meaningfully document changes made. Correct use of .gitignore and Git LFS with Unity. Prefab-based scene construction used to

				avoid merge conflicts in scene edits.
C (70%)	Good work. A clear ER diagram correctly showing the main objects in your game. Some incompleteness in terms of structure and detail.	A testing plan covering all the required features, with detailed instructions and expectations. Bug testing has been conducted. Clear descriptions of bugs with detailed instructions for reproduction.	General compliance with the C# Style Guide, with some minor issues. Code is readable but has some code-smell that needs to be addressed. Code architecture is adequate but could be improved.	Task allocation shows clear allocation of most tasks. Some vagueness in task descriptions. Workload is mostly balanced. Feature-branching version control workflow largely implemented correctly with minor errors. Multiple commits over the life of the project. Commit messages refer to changes made without detail. Correct use of .gitignore with Unity. Prefab-based scene construction used to avoid merge conflicts in scene edits.
D (60%)	Poor. Shows understanding of the format, but significant gaps in the documentation, or discrepancies between document and code.	A testing plan covering most the required features. Some vagueness in instructions and expectations. Bug testing has been conducted. Some vagueness descriptions of bugs and instructions for reproduction.	Significant issues with code quality. Inconsistent application of style. Poor readability with code-smell issues. Code architecture could be improved.	Significant gaps in task allocation. Task descriptions overly brief or otherwise vague. Workload is roughly balanced. Version control workflow significantly deviates from expected workflow. Multiple commits over the life of the project. Commit messages sometimes vague or missing. Poor use of .gitignore.
F (40%)	An attempt made that shows little understanding of the format or purpose of design documentation.	A testing plan that misses significant requirements. General vagueness in instructions and expectations. Little to no evidence that bug testing has	Significant issues with code quality. Inconsistent application of style. Poor readability with code-smell issues. Messy code architecture with significant	Major gaps in task allocation. Task descriptions overly brief or otherwise vague. Workload is poorly balanced. Version control is poorly used with little

		been conducted.	encapsulation violations.	discipline. Only a small number of large commits made. Commit messages often vague or missing. No use of .gitignore.
0%	Nothing submitted	Nothing submitted	Major issues with code quality. No attempt to follow style guide. Major readability problems and code smell throughout. Messy code architecture with significant encapsulation violations.	Task allocation not submitted. Version control not meaningfully used except to submit final game.