Tasks, Aims and Objectives

“Equestrian World” is a blockchain single-player horse racing game, which aims to recreate the real-life riding experience as closely as possible.

The game consists of a few minutes long session (up to 7, but usually around 2-5), where the player jumps over different obstacles in a pre-defined order. A route is created to help him to orient himself on the board. The horse may exhibit unpredicted behaviour, exactly as a real-life one might. The first POV camera is used, capturing the view ahead, the horse’s neck and head, which is exactly what a rider sees in real life.

Blockchain technology is used to verify the horse’s existence and determine its data. It allows the horse to be in the gamer’s true possession and makes each horse unique.

Racing concept is based on cross-country riding: a contest happening on the second day of the equestrian sport competition called eventing. It has relatively complex tasks, which take into account jumping skills, speed and general “cleanness” of riding. In real life eventing is considered one of the most challenging equestrian sports, with cross-country being its hardest part. The game keeps a lot from the original cross-country, particularly the penalty rules, but modifies other things (especially scoring) to make the virtual experience more engaging.

User Instructions

The game consists of short sessions with one ultimate goal: to get the best result possible. The result is measured by three scores: Obstacle Score, Penalty Score and Time Score.

Obstacle Score measures how far the player got in the trek. Each obstacle gives a specific number of points (depending on the difficulty). The points are gained if and only if the obstacles are overjumped in the specific, pre-defined order. A route is generated and provided for the player to make the game process clearer. Obstacle Score is the most important measurement.

Penalty Score measures how “clean” the riding was (how obedient the horse was, whether the player tried to ride too fast) and whether the trek was completed on time. Penalty Score becomes important if two trek ridings yielded identical Obstacle Scores (the most common case: both treks were finished successfully).

Time Score measures how much time the riding took. This is the least important of all three evaluations.

Controls:

- S - slow down
- W - speed up
- A - turn right
- D - turn left
- SPACE - walk

- 2 SPACES - trot
- 3 SPACES - canter
- HOLD SPACE at least 1s before an obstacle - jump

Penalty counting:
- Dangerous (too fast) riding: +25 points on each step
- 1st refusal/run out: +20 points
- 2nd refusal/run out: +40 points
- 3d refusal/run out: elimination
- Time after exceeding the time limit: +seconds*0.4 points
- Exceeding the time limit twice: elimination
- Horse exhausted (no endurance left): elimination

Major Techniques, Algorithms and Tools

I. Animation

The animation was created in Blender. For the sake of simplicity only the horse's head and neck were built, since it is enough to give an authentic game experience (that is what a rider primarily sees in real life) without wasting resources. The animations for all gaits (standing, walking, trotting, cantering, jumping) are different, though similar. Together with the First POV camera they were used to recreate the real life experience as closely as possible.

II. Trek Auto Generation

Each trek in the game is auto-generated (randomly) and unique. The generation consists from the following steps:

1. To tile the area with ground and build the hedge wall around.
2. Create a pre-defined number of obstacles (N=5 in the built game instance). For this:
   2.1. Choose N random tiles of ground away from the hedge; this guarantees that no obstacles will be too close to the hedge, and the route will not be too straightforward.
   2.2. Define height, width, requiredSpeedMin, requiredSpeedMax, requiredEndurance, trust, skittishness, obedience for each obstacle: randomly, but from certain intervals which allow the game to be balanced. Depending on these parameters, calculate the obstacle's score.
   2.3. Choose a random location in the inner part of the chosen tiles (½ of tile size away from each edge); this guarantees that no obstacles will be ever clustered together.
   2.4. Instantiate each obstacle and rotate it at a random angle in [0,90) degrees interval, set the rest of the attributes an obstacle needs.
3. Build the route: for each obstacle, from i=1 to N, find points on the obstacle's normal and reasonably away (pre-defined) from both sides of it. These are the parts of the route where the horse prepares for the jump or runs immediately after. Connect these points with the start and each other in the correct order with colourful lines (the colour is determined by the obstacle position).
4. Calculate and set the limit time for the route depending on its length.

5. When creating the horse at the start point, align it to the route's first leg.

Each obstacle's id, position, height, rotation and data is saved to a file, which allows to recreate this trek during another session. The number of tries to ride through the trek is counted and displayed on the game start screen.

III. Horse Parent Object

The Horse Parent Object fosters all objects connected to the horse, which are:

- Main Camera (to provide the 1st POV).
- Top Camera (to make orienting easier for the player).
- SphereForTopCamera, a bright shadowless golden sphere above HorseAnim, invisible from the Main Camera, but clearly pointing to the current HorseAnim location for the top one.
- HorseAnim, main object with horse animations and scripts.
- Rider, its child, which is responsible for the controls.

IV. Horse AI

One of the game's main aims was to simulate a realistic behaviour of a horse. This was achieved with a detailed AI. The most important features were defined as follows:

- A horse can change his gait, start turning, change current speed only in the beginning of a new step, since no real-life horse can do these things in a not stable enough position (e.g. in the middle of a leap phase of a gait). The balancing challenge was to define the appropriate step length and speed for each gait, as well as the number of degrees a horse would realistically turn during a step.
- The horse slows down during turning.
- The horse will never run on an obstacle or a hedge. This was ensured by the raycasting algorithms discussed in class.
- There are three types of misbehaving: stopping, running out (turning randomly and running away for some time, uncontrollable by a rider) and spooking (same as running out, but even longer).
- The horse will misbehave if he receives simultaneous (during the same step) commands to speed up and slow down, since he does not understand which command to obey and is confused.
- The horse will not jump if there is no obstacle ahead.
- The horse will adjust his speed before an obstacle to stop or jump over it (and to ensure he will not run into it).
- If there is an obstacle ahead, but there is no command to jump, the horse will simply stop.
- If there is an obstacle ahead, and there is the command to jump, the horse has two options: to obey and disobey. The algorithm is as follows:
    1. He checks whether the leap will be safe: if the obstacle is not too high or wide, the current speed falls inside the safe interval for this specific obstacle and he has enough energy (endurance) to overjump it. If the leap is too risky, i.e. the rider urges the horse to do something dangerous, the trust of the horse will drop.

2. Even if the jump is potentially safe, the horse may be too scared to undertake it. To determine this, the parameters of skittishness, trust and obedience are compared with those required by the specific obstacle. If the horse is indeed scared, his skittishness becomes even greater.
3. Even if scared, the horse may still jump, if generally he is obedient, trusting and not too skittish.
4. Illogical like every living creature, the horse, even safe and not scared, may still disobey. This is determined by random, but depends on general trust and obedience as well.
5. If the horse overcomes the obstacle successfully, his trust increases, while skittishness decreases.

- If the horse disobeys, his obedience drops, so the probability to overjump other obstacles lowers as well.
- After a jump the horse canters for some time, no matter the gait before it.

V. Blockchain Technology

A horse is created as an asset in an Enjin project. The game connects to the platform once, during the first trek launch, to obtain the data of this asset, relates the name of the asset with the files it already has and, if this operation is successful, creates a proper horse object.

VI. Events

All communication of the game entities is done via events. They can be roughly separated into a few groups: obstacle, trek riding and game management related.

The obstacle related group consists of ClosestObstacleFoundEvent, NoObstacleDetectedEvent, ObstacleSuccessEvent, OverjumpedObstacleEvent, RefusalEvent, RunOutEvent. They are fired by the Horse, to notify the rest of the program of its current state. Mostly used by GUI scripts to show this data to the player.

The trek riding related group consists of DangerousRidingEvent, FinishEvent, HorseExaustedEvent, RefusalEleminationEvent, TimeRanOutEvent, SimpleRefusalEvent. They are also fired mostly by the Horse, but, in contrast to the previous group, bear heavy influence on the further riding through the trek (or lack thereof).

The game management related group consists of HorseConfigFileIdentifiedEvent, SceneFinishedEvent, TrekCreatedEvent. These events are required to regulate the correct and efficient loading of the gameplay scene or signal the need to go to another one.

VII. Interface

The interface of the start and finish scenes consists of a panel, text with the best or the last results of the trek, and buttons to close the application or go to another scene. Pause and help screens are also available.

The main start menu allows the player to go to a new trek or to continue improving the last one. It stores the best results, which makes comparing and seeing the progress easy for the player.

The gameplay scene has a few panels: the very top one shows current game scores, panels on the left display the data about the horse's movement and the closest obstacle (if one has been detected). They are needed to prepare for a jump. The panel on the right demonstrates the list of the obstacles in the route, together with their scores and statuses (whether it was overcome successfully). It is required to follow the progress.

Specification of the source code

I. **Game Manager class**: regulates the beginning and ending of the gameplay scene. It instantiates the trek, then the horse, then the HUD, always waiting for the creation of the previous item before proceeding to the next one. It also loads the Finish Scene after the session is over.

II. **Trek-related classes**:
- Trek - instantiates a trek as discussed above or recreates it from saved data, controls the horse's progress, notifies of the completion of the route, saves the results.
- TrekIO - an extension of the Trek class which holds all input and output functions (i.e. those that are connected to work with files).
- TrekTiming - implements the timing system: whether the pre-set has been exceeded, and if yes, whether it has not been exceeded twice. Uses the utility class Timer.
- TrekStaticData - a static class that holds data related to the trek (such as scores, difficulty, trek status (loaded/newly created), the next obstacle number etc.) which must be accessible from a number of other classes and objects.

III. **Obstacle class**: holds the data about the obstacle. Provided with the horse's current physical state (speed, endurance, jump abilities) determines whether he is qualified to jump over, and provided with the horse's psychological state (trust, obedience, skittishness) decides whether it is agreeable for the horse to jump over.

IV. **Rider class**: gives commands to the horse. This class is responsible for the controls.

V. **Horse class**: implements the horse's AI. More details about it can be found above. It also initiates and uses a gait manager to control the movement. This is the most complex and important class of the game.

This class has a few groups of attributes:
- Next step related: the next gait, whether there will be necessary to turn, slow down, speed up.
- Jump related: speeds for jumping, where the target to jump to is placed etc.
- General data about the horse's abilities, such as how far he looks to notice the obstacle or how sharp the possible turns may be.
- Data about the horse's soft skills: trust, endurance, obedience, skittishness.

- Others (e.g. gaitManager).

This class may be roughly equivalent to a horse's brain.

VI. **Gait Manager & Gait classes** (Jump, Canter, Trot, Walk, Standing): implement the movement. They control speed, gait type and animations. They are not responsible for making any decisions, only for implementing what has been commanded by the horse class or signal if it is impossible to do.

This class can be roughly equivalent to a horse's muscles.

VII. **Event Manager & Events**: as explained above, events play a crucial role in making the code efficient, low-coupled and robust. All events go through a static Event Manager, which stores invokers and listeners and matches them.

VIII. **Data Parsers & Utils**: since most of the important data is stored in files, a few parsers and static utils were created to separate the task of parsing from main gameplay classes. They are mostly stored in the ConfigData folder.

IX. **Interface and Menu-related classes**: short classes that update the interface boards, keep track and update the scores, load other scenes or screens. Can be found in the Interface and Menu directories.

Performance and Optimization

1. **Object handling**: all objects present during the given scene are necessary in it and destroyed afterwards.
2. **Code structure**: the code was written to avoid duplications and for each part to be executed as few times as possible. The events are used extensively for better encapsulation, lower coupling, avoiding time- and memory-consuming operations such as search etc.
3. **File input and output**: most of the data (horse specs, trek specs, results) is stored in files. The amount of information is low and is expected to stay so even with further scaling. Hence the decision to avoid threading and handle all files' input and output in the main programme: in most cases the programme must wait until the writing is completed to be executed correctly, or waits for another event (e.g. end of countdown). So the bottleneck caused by writing to the files is not noticeable for the user. By virtue of low data amount it is fully expected to stay this way and be preferable to the threading tradeoffs.
4. **Blockchain**: the game connects to the Enjin platform only once, to retrieve the necessary data (the horse's name, which defines the name of the file to be used) and store it in the static data. This allows to avoid multiple connections, wasteful from the point of time, resources and gas money (KETH).

References

1. Code: no ready code pieces besides those discussed in the classroom, official tutorials found on enjin.io, Unity and docs.microsoft.com C# documentation were used. The game was written from scratch.
2. The textures of grass, hedge and obstacles were taken from https://ambientcg.com . The texture for the horse was created in Blender from scratch.
3. The horse's behaviour was defined from personal experience.
4. The picture for the Main Menu Screen was taken from https://equestrianco.com
5. The rules for the cross-country riding were taken from https://www.badminton-horse.co.uk/eventing-explained/