Caviators

A game by Arbitrary Team Name Productions

Markus Webel
Tim Türk
Domenik Popfinger
Daniel Hook

For

TUM - Computer Games Laboratory
Winter Term 2017/2018

Game Proposal

1. Game Description

"Caviators" is a 2D singleplayer arcade game where the player controls two cavemen, gliding down a cliff to catch birds. Since they cannot stretch the net open by themselves, they have to do it "Together", which was also the course theme for this game. The difficulty of the game lies within controlling two characters at once. Every move changes speed, direction and the opening and shape of the net, which makes it a necessity let both characters to work together to achieve the higher goal: Catching birds and keeping them.

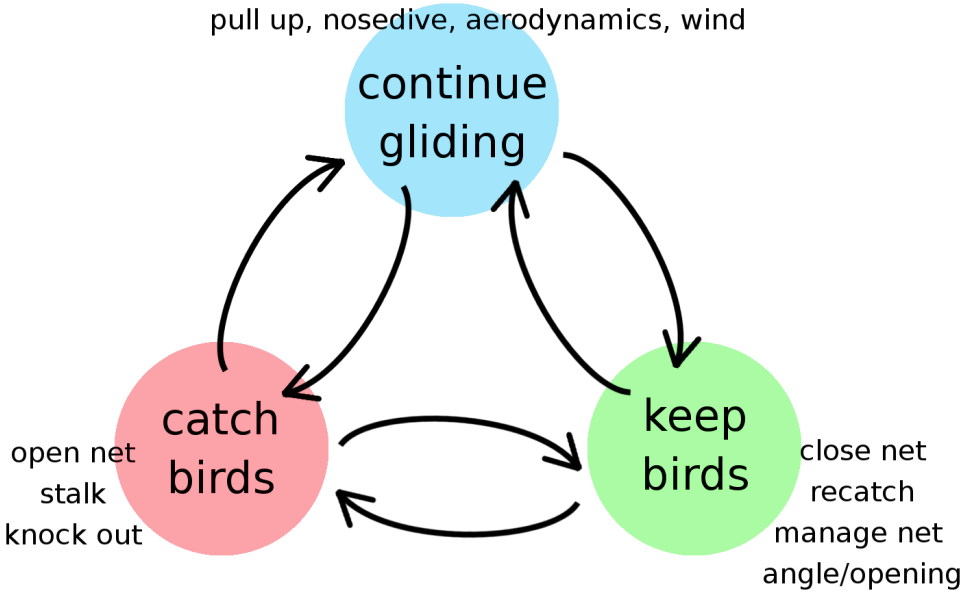
It is inspired by the arcade gameplay of games like "Luftrausers" and the controlling of two characters as in "Brothers – A Tale of Two Sons".

2. Gameplay

The game is split into two phases: First, the gliding phase which lasts about 1-2 minutes after each jump. Second, there is an upgrade phase afterwards, where players can buy enhancements to catch even more birds in the next round.

2.1 Gliding phase

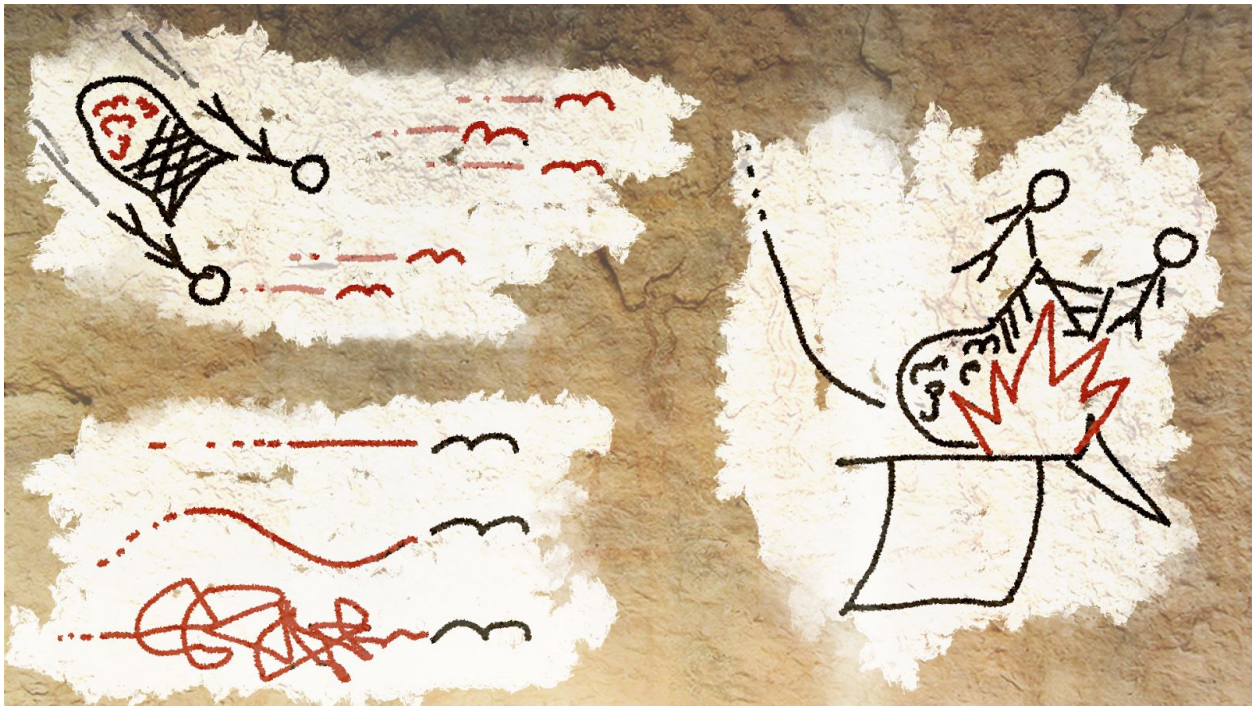
The gliding phase takes place in the same level every time but with differently spawned birds and wind zones.
Gliding can be split into three core mechanics, the player has to balance:



2.1.1. Continue gliding

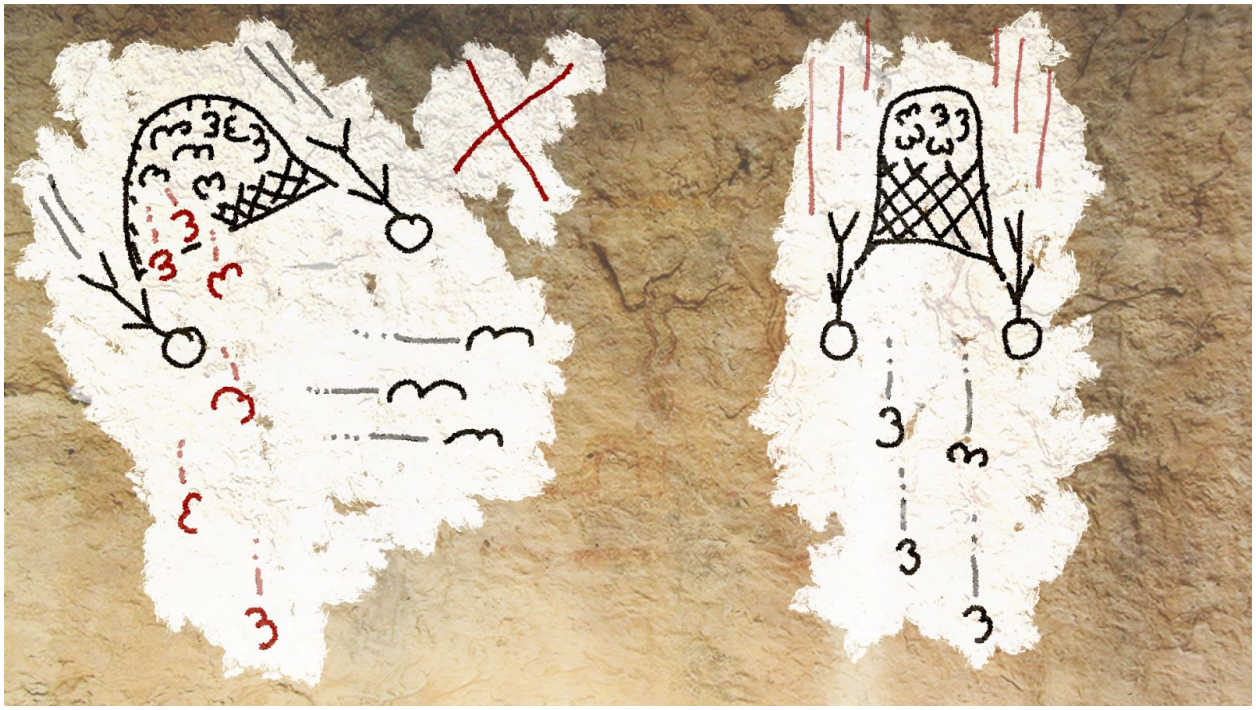
As with real physics, the cavemen drop over time. But by using troll physics, they can pull each other up again, by pulling the net. Another method is to fly into wind zones that pull the cavemen back into the air. Also they can nosedive downwards to generate speed, which lets them fly even higher up, due to troll physics. Another way to generate speed is by enhancing their aerodynamics by flying close together. To not bring a fast stop to the travel, they have to avoid crashing into big dinosaur birds.
Staying in the air influences the other two goals significantly: If the cavemen fly near together, it is harder to catch birds with the net between them. If they have to change their path to reach a wind zone or avoid big birds, it is harder to catch birds. A nosedive can easily lead to bird droping out of the net. If the player ignores gliding and drops too fast, the round is over.

2.1.2. Catch birds



The main goal is to catch as many birds as possible. Birds move in different ways and groups, which makes it harder to catch them. Also there are big birds which have to be knocked out with a net full of small birds, so they do not fidget in your net.
To catch birds there is always the balance between opening the net far enough to get them, and keeping it closed enough the prey does not drop out. Since flying is harder as the net gets filled, the game naturally draws to an end.

2.1.3. Keep birds



Your net is open, so birds can always drop out. To keep them inside, the player has to balance the size and angle of the opening of the net in relation to his current velocity.
If birds drop out of the net, catching them before they hit the ground becomes the main priority, making it a lot harder to keep yourself in the air or catch new birds.

2.2 Upgrade phase

During the upgrade phase, the player can buy upgrades which are paid for with caught birds. The upgrades give the player an overarching goal, and also change the gameplay every round. To balance the system, the player can only activate certain upgrades and every advantage comes with a disadvantage, leaving the player free to adapt the game to her taste, but not becoming overpowered.
Upgrades mostly change the flight physics (e.g. attached wings), the net (e.g. bigger net, sticky net) and the spawning (e.g. changing wind zones).

3. Technical Achievement

The game will be implemented using the Unity game engine. Due to the previous experience of the team members with the engine, the focus can completely be set on the gameplay features.

3.1 Cave painting shader

The main technical achievement is a cave painting shader, that renders the whole game as if it was drawn on cave walls, while still remaining readable for the player. The whole scope of the shader has to be tested, but there are a lot of directions to go to: E.g. generating and endless, non-repeating cave wall by combining texture chunks or using motion blur to generate the impression as if every previous frame was wiped away, as seen in chalkboard animations.

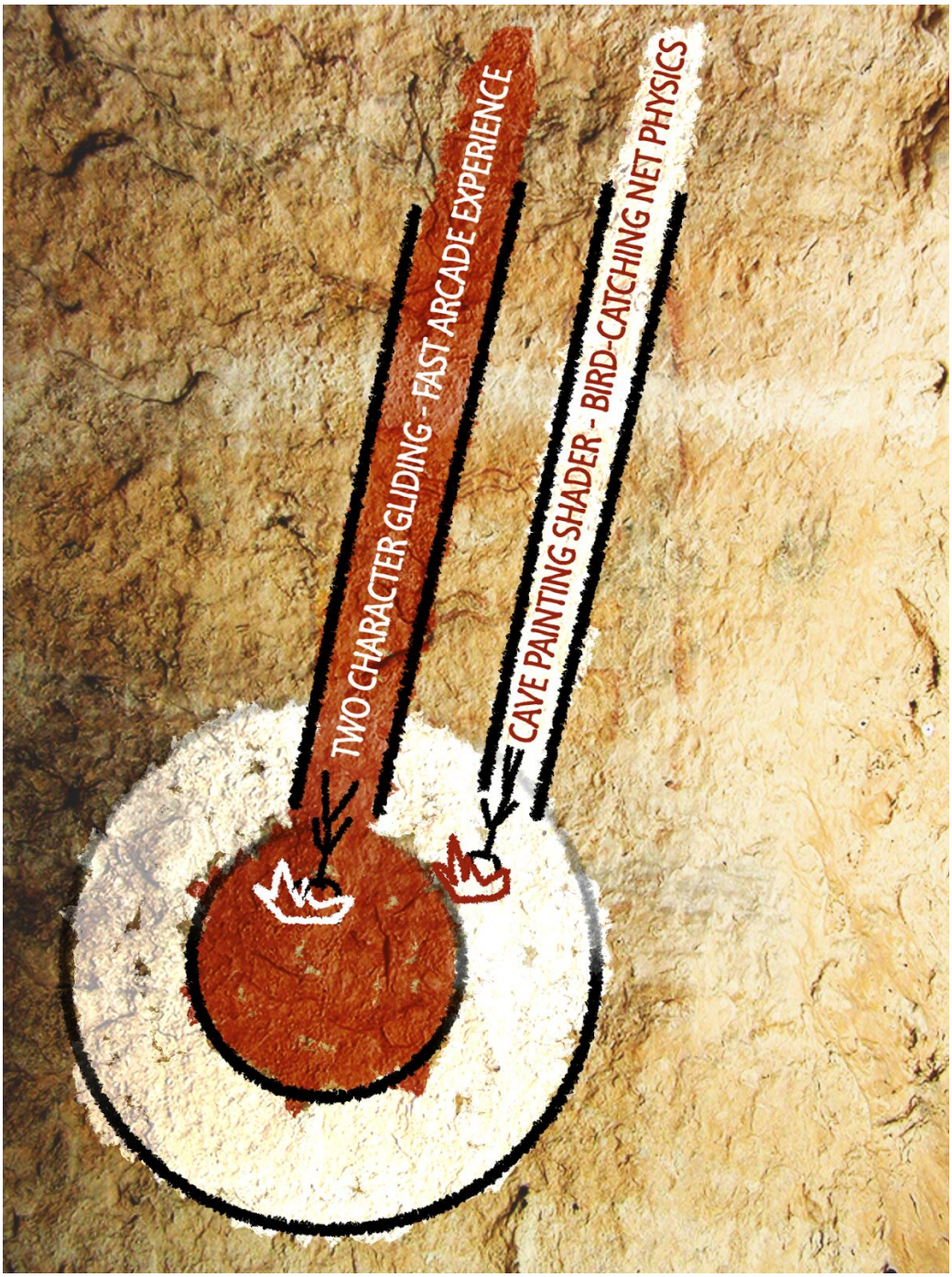
3.2 Flying and net physics

As our main mechanic, the physics have to fulfill a lot requirements: The path of flight has to be easily controllable and support maneuvers like nosediving followed by pulling straight up to gain height and speed. The net physics has to keep the birds inside the net, as well as leaving the chance for birds to drop out. Also the net has to be controllable enough to knock out big birds. Aside from the game design aspects, the physics also have to look reasonable to an extend and feel good. Achieving this requires a lot of technical tweaks and systems.

3.3 Extras

If there is enough time, we would like to try a wind grid instead of wind zones. Therefore we would simulate the wind movement in Mantaflow and import it to our Unity level.
Another interesting feature would be to implement 3D cloud shading, and projecting it then onto our cave wall.

4. Big Idea Bullseye



5. Development Schedule

Below is a sorted layered feature list. Since the core gameplay is very compressed and interdependent, most higher layered goals are mostly about effects, graphics and adding gameplay tweaks.

5.1 Layer plan

Functional minimum:
- Flying physics
- Net physics
- Birds
- Score screen

Low target:
- Cave painting shader 1 (drawn-like style)
- Upgrades 1 (longer gliding, bigger net, ...)
- Tutorial levels
- Basic sound effects (flying, hitting birds, ...)
- Simple menu
- Feedback effects (camera look-ahead, screenshake, ...)
- Static wind zones

Desirable target:
- Cave painting shader 2 (motion blur)
- Upgrades 2 (sticky net, slaves that create wind fields, ...)
- Different bird types (different sizes, different movement, ...)
- Menu in cave style
- Generated levels
- Gameplay effects (birds loosing feathers if hit, ...)
- Assets for Upgrades
- Grid based wind zones

High target:
- Fluid sim to generate wind fields
- Cloud shader (obviously drawn clouds)
- Story screens
- Cavemen voice acting
- Environment art (trees, ...)

Extras:
- Worldmap
- More bird types
- Different maps (winter, summer, other topology, ...)
- Getting bought by Tencent

5.2 Timeline

See attachments.

Colors:
Orange: Markus
Blue: Domenik
Purple: Tim
Green: Daniel

6. Assessments

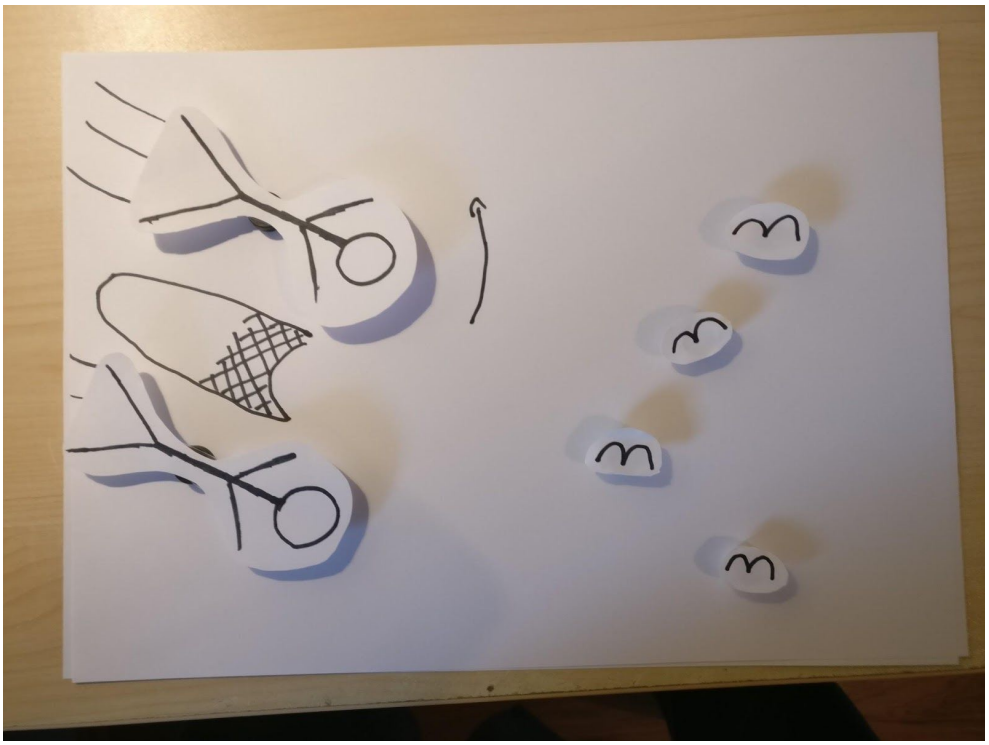
"Caviators" is intended to give the player a fast-paced arcade-like flying experience. While the basic experience should just be a cool flying controller that provides fun short rounds of gameplay, there is also another goal: Fluidly balancing all three core mechanics under the strict time pressure of the approaching ground. Therefore it is crucial that the player is able to control both characters without being confused and gain experience to master her flying skills. Via upgrades and different level spawning there should be enough variety to always start another round, as well as the possibility to shape the gameplay in favor of the player's taste. Also the player should be able to set her own goals, like catching specific birds to work towards a certain upgrade or beating a highscore.
In the end, it should also be a humorous experience. Let's be real, it's a game about two falling cavemen.

Paper Prototype

1. Design

When starting out with designing our paper prototype none of us could formalize how to capture the speedy and fun flight experience we imagined as core to our game idea.
So we went about creating a paper version of our game by putting a game master in charge of modelling physics. This game master (in the following “persona computa” or “PC”) would periodically ask for our player’s input, and then model the reaction of our cavemen-gliders and their environment, resulting in a turn based interaction with the following elements:

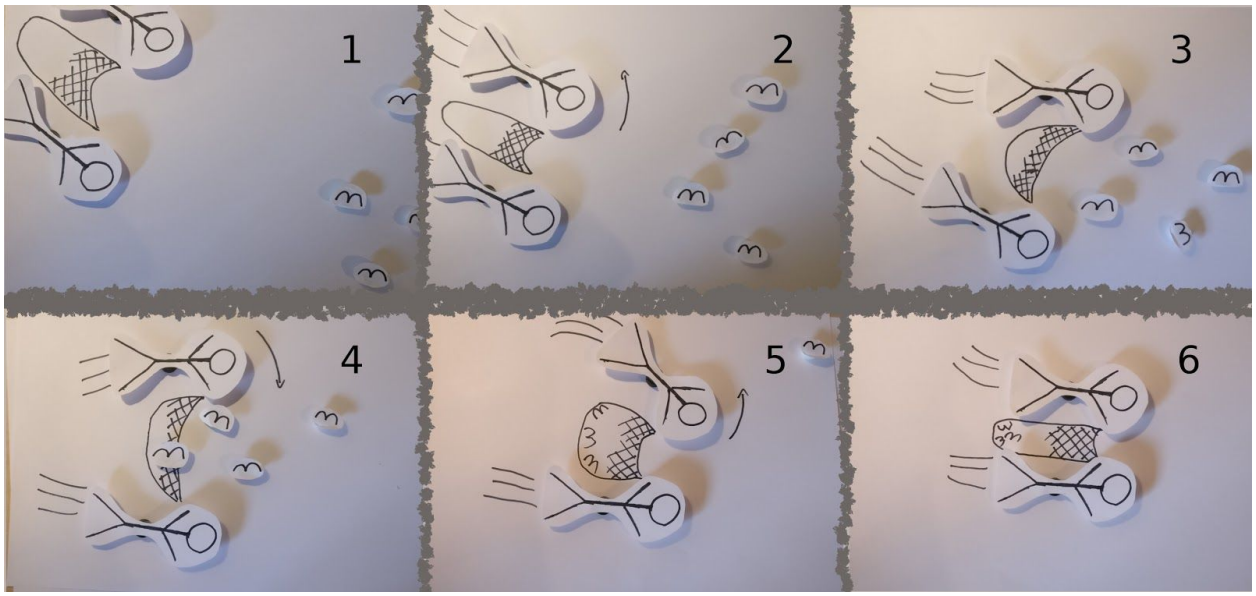
- 2 caveman-gliders
- 1 net (different versions depending on how far it’s opened)
- Either several small birds, or a single big one.



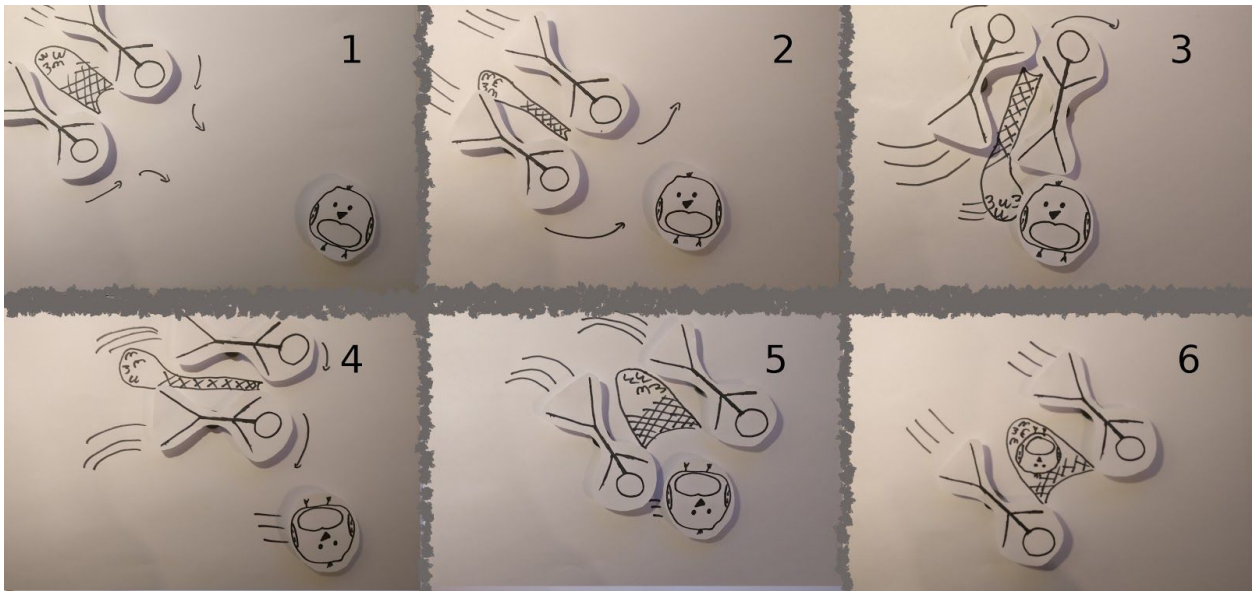
In practice that meant the player controlled whether each caveman would turn their nose upwards, (to gain some height in exchange for speed,) downwards, (gaining speed but losing height,) or continue their flight as in the previous turn.
In turn, the PC would then move the cavemen depending on the player’s input and their own understanding of flight physics, set the net between them depending on the cavemen’s positions, and finally move all available birds - either into the net, or so they would continue flying.

2. Experience

To get an idea of how the prototype works, here are visual transcripts of two rounds we played:



This first one shows the standard game situation in which the player catches several small birds. Between each pair of the six shown key turns, there were one to three game turns in which the player could give input.



In this second situation the player already has caught a couple small birds, and now chases after a fat bird in an attempt to knock it unconscious and then catch it.

2.1. Problems

During playtesting the prototype, the split of player and PC led to some quarrels about the flight behavior. The PC could not always provide an experience that was consistent with our player's expectations: the reaction rate of gliders would differ as well as the timestep size and the swing speed of the net.
Another problem was that we already had put a lot of thought into our design, meaning the PC would move the player characters in ways so that our balance triangle would always work out.
Also the net proved to be hard to represent with a physical prototype. Using a string seemed to be the most obvious solution at first, but the strings we had were either too stiff or too thin. Bringing them into positions resembling those of a net at high speeds (and keeping them there) was annoying.
Overall. the paper prototype did not convey the sense of speed and urgency that's core to an arcade-like game, so a lot of the experience could not be tested with the paper prototype.

2.2. Conclusions

Although this playtesting was not an acceptable recreation of the intended gameplay experience, it led to interesting discussions about other aspects of the game:
The disagreements between players and PCs about things like glider and net behavior showed us some actions that players wanted or expected to be able to do, even though they were physically incorrect behavior. For example players tried to pull up and slow down with one glider, in an effort to swiftly swing the net forward. The intended result could not happen unless the net's center of mass was unrealistically close to the gliders (or unless that action was accounted for in our game's physics computations.)
Another problem we already had thought about but did not grasp the scope of so far, was the problem with loops in the net. If players were to cross paths, then a two dimensional net would form a loop around our caught birds, and would prevent us from catching new birds.
Here the paper prototype helped us visualise the problem, as well as our ideas for solutions, so after a lot of discussion and explanations involving some sort of puppet theater, we agreed to use the convex hull of a simulated net to ensure that no loops would form.

3. Results

In the end, we did arrive at some points that we could incorporate into our game design plans:
The probably most important one was the decision to put more focus on fun interactions than on realistic physics. So when implementing the net, we'll aim to not only simulate it as if it was dragged along by the gliders, but that we'll try to anticipate how and when the players want their net to open, close, or swing around, and that we'll try to enable that behavior.

For the gliders themselves this means that we'll explore a wider variety of control schemes and behaviors, and that we won't discard approaches or ideas just because they sound unrealistic. Instead we'll see which combination of controls and flight behavior best supports the interaction patterns we expect, and those we have seen emerge in our prototype.

Implementation specific points we discussed while playing the paper prototype for example were disabling player-player, and player-net collisions. While our game will be 2D, we agreed to go about implementing it as if one glider always was behind the other on the third axis.
If communicated visually, this would result in game situations in which one glider passes through the other in 2D not appearing confusing to the player, while also giving the player more freedom in how they control the gliders.
Additionally, this would allow us to place the net between the two gliders, resulting in a visual representation in which it makes sense that the net never wraps around itself.

Lastly, in one session with the paper prototype we had the problem that an unconscious big bird was falling downwards directly below the gliders, which meant that the player could not catch it. It would have taken too long to first slow down to get behind the bird, before going into a nose dive to chase after it.
After we started giving big birds a boost when they were hit with a filled net this problem was resolved, so we'll incorporate such a boost when implement batting big birds out of the sky.

Interim Report

Work on our initial prototype was mostly split into separate components, so each of us would be able to focus on few, more specialized problems, and to work on them on our own times. These separate processes and their results are outlined in the following chapters.

1. Flight Controls

With flight controls being the way players would interact with our game, we had to immediately put together a rudimentary implementation that we could build on, and test other modules with.

As was expected, working with that initial implementation showed a plethora of points to improve on, both to make the game more fun, and to make developing it easier. Armed with that knowledge we wrote a second, more thought out implementation of flight controls while the other modules were being developed based on the first version.

1.1. Paleolithic Control Scheme

The goal for this version was simple: make an object behave as one would expect a glider to. In pursuit of that, we first experimented with fully physics based controls.
The most important advantage of such an approach over directly controlling the glider's position, orientation, and speed was that it allowed us to later add other control forces (for example from collisions with big birds) without having to add convoluted case distinctions in our control code. Sadly, the glaring disadvantage of this glider implementation was that it handled like a rock, so we changed our approach from simulating glider physics, to simulating only their desired effects.

The result, our initial flight controls, for example simulated air resistance by reducing the glider's velocity based on estimated drag, and by forcing the glider to turn towards where it moves.
(A behavior resulting from having minimal air resistance while aligned with the direction of movement. Just like boats would turn when being dragged through the water sideways.)

For the first playable version of our game, two such gliders were connected to a simple sphere via standard Unity spring joints. An edge collider in between acted as our net, and our planned mechanic of gaining momentum by pulling one glider towards the other was implemented by at the press of a button adding impulses with fixed strengths to both gliders.

Testing with that version showed much we needed to improve in the next one: for example turning needed to feel more responsive, and the small, still mostly realistic amount of height one could gain from pulling up after a nosedive to build speed felt underwhelming. These properties also were difficult to adjust, because the initial code structure didn't provide clear control valves.

1.2. Neolithic Control Scheme

Rebuilding our flight controls from the ground up had one main goal: to present parameters in such a way that it would become possible to predict what changes were necessary, based on a vague description of the desired behavior. Ideally, it should become possible to formulate implementation strategies for ideas like “in situation x the input y should lead to a little more z.”

To that end, a fuzzy logic inspired approach was chosen: a data gathering and preprocessing script exposes information that a classification script uses to rank the glider’s movement in terms of similarity to select archetypes like a nosedive, climbing, or gliding.
Both preprocessed data and the classifications are then remapped and used as parameters for the actual flight controls. With that setup the control scheme can be changed by adjusting the curves of remapping functions in the Unity Inspector, or by editing reserved control factors.

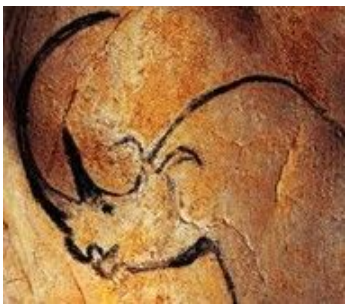
This much clearer code structure already helped in producing more responsive controls, and while we as of now e.g. haven’t implemented a fix for the underwhelming height gains problem, its problem description now directly leads to an implementation strategy like: “gain momentum while nosediving, exaggerate momentum to boost height gained while climbing.”

2. Cave Painting Shader

It was obvious from the beginning that the cave painting shader should be implemented as a screen-space effect. The main motivation was that a certain ‘wiggliness’ to the cave drawings was desired, yet such an effect is hard to implement on a per-object basis because of the differences in mesh density and scale. Tessellation would have been an option, but a screen-space effect seemed much more reasonable. Other benefits include:

- Independence of any lighting system
- No distinction between sprites (e.g. menu options, text) and 3D geometry
- Level designers can easily switch the shader off
- Centralized customization vs. per-object settings

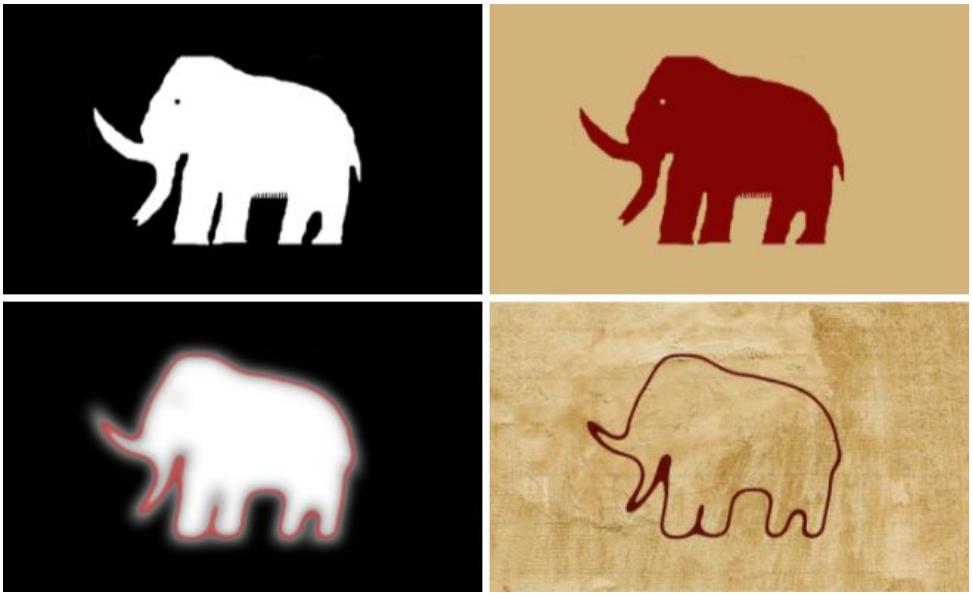
2.1 Prototype: Blur-based outline shading



Inspired by the outline-based drawings in some caves, our first idea was to recreate that style by rendering outlines of all objects on screen, using the colors from the original object so it would still be possible to differentiate the two player characters from the background and from each other.

The first shader prototype implemented this by blurring the screen buffer alpha channel using a 5-pass Kawase filter, reapplying their

original colors to the thresholded alpha values, and finally combining the outlines with a background cave wall texture. Some of the intermediate steps are shown below:



In the end, we decided against using this approach because of issues like:

- The loss of detail around edges of the objects (see: mammoth tail)
- The complete loss of detail within the outlines (see: eye of the mammoth)
- Problems with differentiation between objects despite outline colors
- Problems with the handling of semi-transparent objects
- Uneven thickness of the outlines

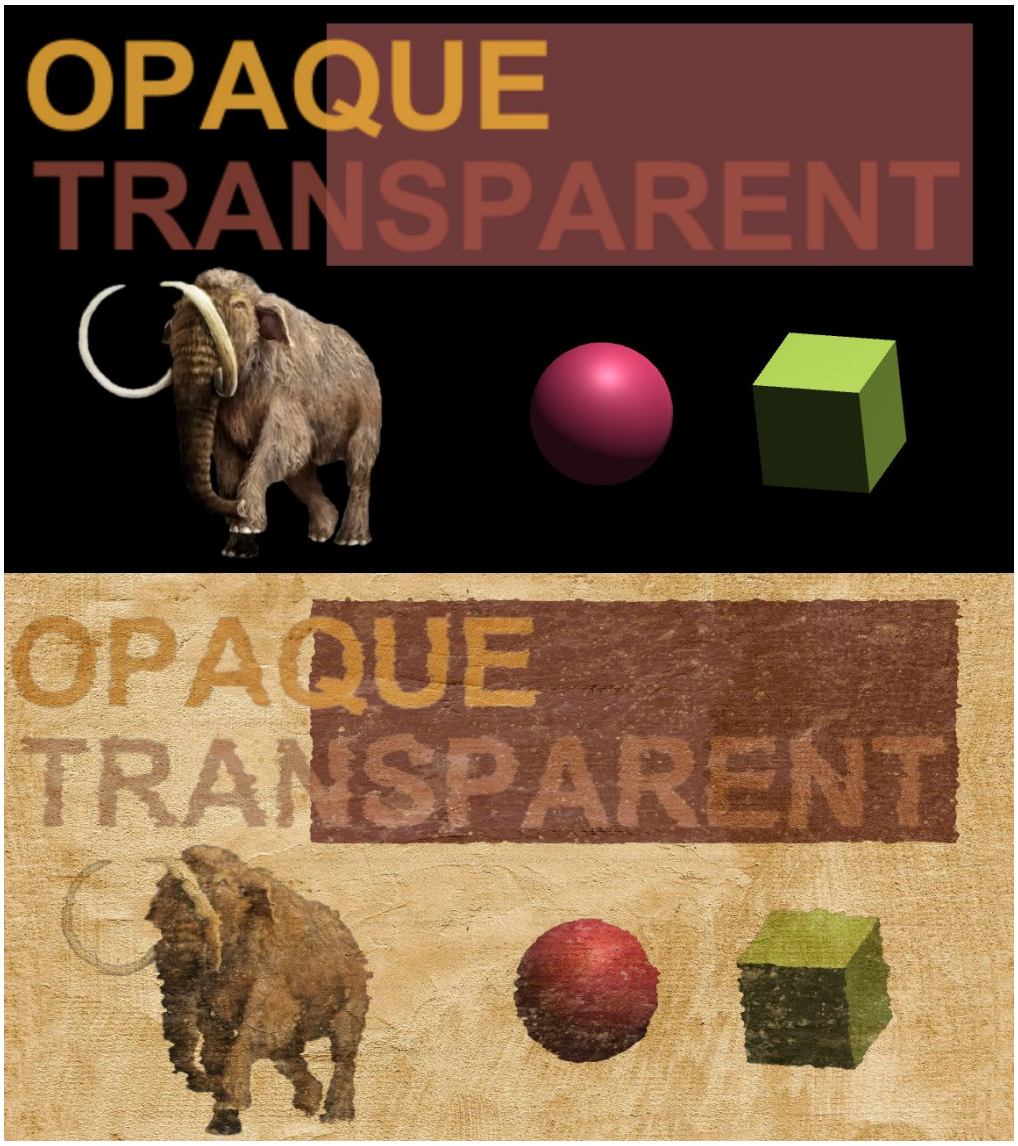
2.2 Final shader: Alpha-based linear interpolation

To address the aforementioned issues, we abandoned the outline-based stylization of the scene in favor of a more traditional fill-based approach, which historically was used by not only the cavemen, but the cavewomen and the cavechildren too. This made the resulting shading process much simpler and significantly boosted performance.

The new algorithm e.g. uses paint textures to show object fills on a cave wall texture, simulates motion by computing texture-space offsets from the camera position, and introduces additional offsets which creates wiggliness. Its effects can be controlled by exposed shader variables:

- Compositing weights and artificial filling
- Wiggliness frequency and intensity
- Background and paint textures

This version of the cave painting shader performed adequately for our purposes:



3. Net Physics and Rendering

We decided to try our luck with the built-in Unity physics joints instead of implementing net-physics from scratch. This approach has some advantages:

- Collision handling is done by Unity in the correct time step
- Unity can interpolate the positions of physics objects between physics timesteps
- Mass, friction and bounciness are available via built-in settings
- It runs natively on all target platforms, as opposed to a naive managed implementation

3.1 Chain-Link Implementation

The net is implemented as a sequence of invisible mass points connected by distance joints. Distance joints apply a contracting force when stretched beyond a specified length, thus simulating stretching forces within the ropes of the net. This is not particularly realistic, but makes for more interesting visuals and gameplay.

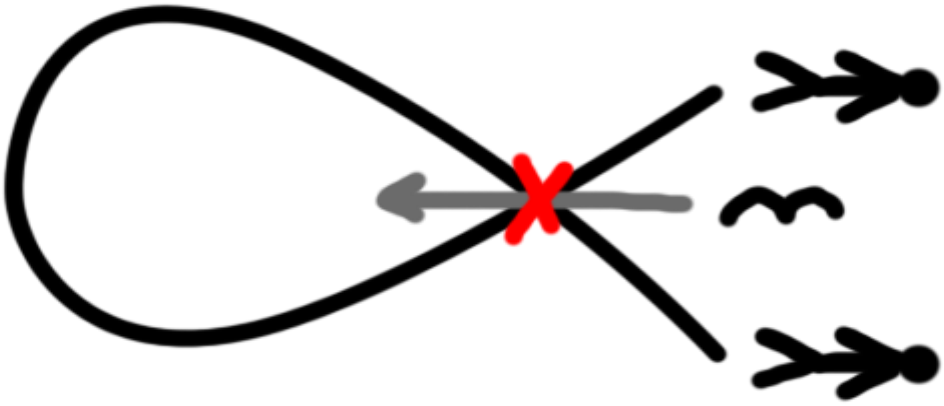
The mass points are created from a single prefab, allowing for quick prototyping of net physics and for replaceable nets at runtime. Among the exposed properties are:

- Mass-point mass (also controls stiffness of the net)
- Contracting force strength
- Number of mass-points (affects performance)
- Contracting force threshold distance

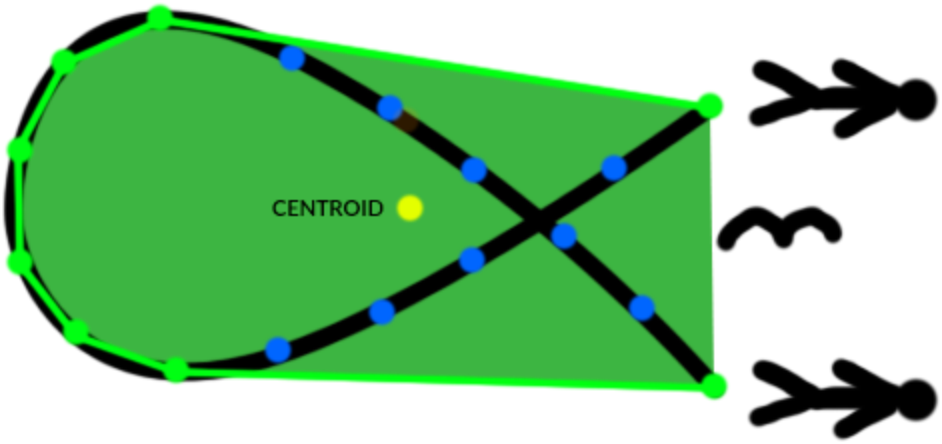
After testing various setups, we adjusted the collision settings to avoid all collisions between net components, between players, and between the two. This worked best for our type of gameplay.

3.2 Collider for Bird-catching

The implementation above worked fine for simulating the inertia and collisions of the net, but was not suitable for catching birds. It also was not particularly visually pleasing:



This is an issue, as we allow both cavemen to cross paths arbitrarily, but want them to be able to catch birds regardless of their spatial configuration. After much discussion we agreed to use an incomplete convex hull as both collider of the net and as base for the visual representation.



Both the Quickhull and Monotone-Chain-Hull implementations were tested. Quickhull performed poorly because of floating-point issues and because the net often was close to a perfect convex hull already, which causes Quickhull to go into deep recursion, resulting in almost quadratic complexity. The Monotone-Chain algorithm performed adequately.

Since most convex hull algorithms give the hull vertices without any particular order, they have to be sorted around their centroid (see image, yellow dot). After this, the positions of the vertices which are closest to the two cavemen have to be determined. The reason for this is that for some rare net configurations, the net vertices that are attached to the cavemen are not actually part of the convex hull, and are culled by the hull algorithm. Again, some odd net configurations can cause trouble by occluding the net opening. The final list of hull vertices uses a simple heuristic to guess the position of the opening, which appears to be sufficient for normal gameplay.

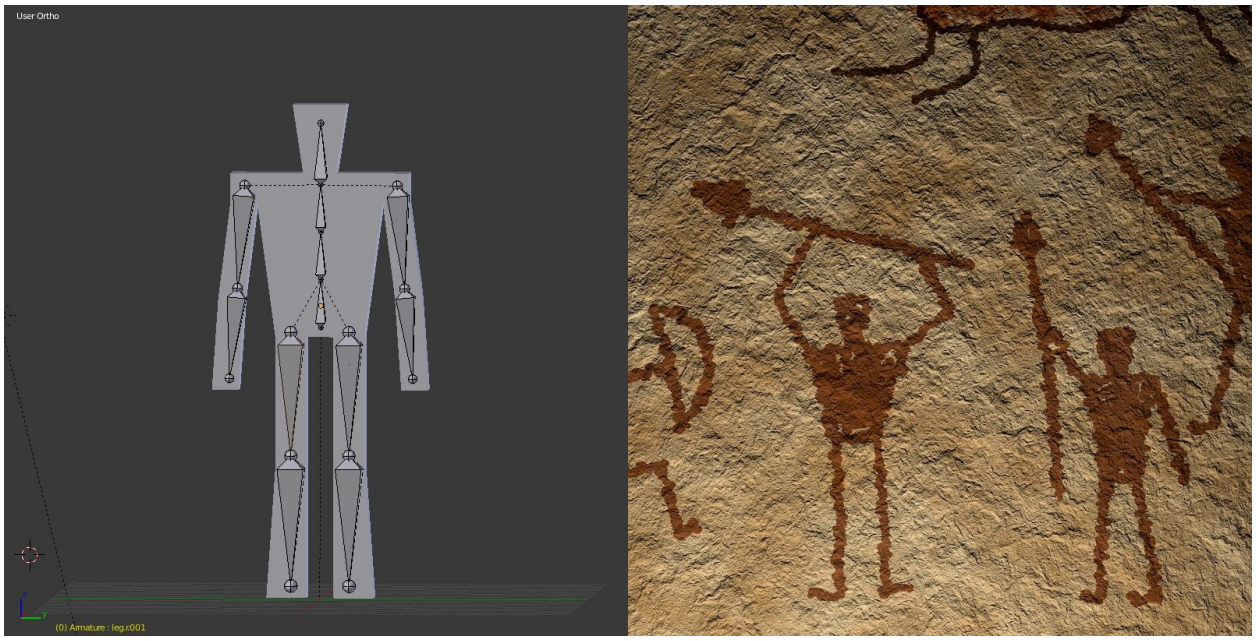
3.3 Rendering the Net

Using the hull vertices obtained before, a simple mesh can be created using the centroid as the ‘tip’ of each triangle. Since the vertex set is a convex hull, this process is guaranteed not to produce any overlapping triangles.

The line spanned by the hull vertices is renderer by a Unity LineRenderer, which supports the inclusion of artificial corner and cap vertices, resulting in a smoother outline. The net is filled with a special shader, which ensures that the center of the opening has a fixed texture coordinate. This is meant to give visual consistency over multiple frames.

Nonetheless, the net fill shader is still a work in progress. We might try to add sensible texture coordinates and some artificial vertices to the generated net mesh, allowing us to use a seamless net texture, which has undergone a polar transformation.

4. Models



Caveman model (+ bones in Blender) next to inspiration from an internet artist's interpretation (https://www.tes.com/lessons/xf8ZwNfgzZG31Q/cave-paintings 3. Dez 2017, excerpt)

The caveman models should look like stick figures, resembling cave paintings. While actual cave paintings usually show really thin limbs, we went for the style of an artist' interpretation of cave painting (see Figure). With wider limbs and a thick chest, the character is easier to see, even if it is only displayed small on the screen. Also the wide chest gives the opportunity to visibly turn the caveman which is an extra opportunity to display gameplay mechanics (e.g. the caveman can be turned if the player changes direction, giving immediate feedback.)
We choose to use 3D models instead of sprites, since we already had some experience with 3D animation, as well as it giving us more to work with in the shader (e.g. vertex shader for effects).

5. Effects

Visual effects in Caviators are very important to give gameplay feedback as well as providing a feeling of speed. Every effect comes with a boolean to turn it off, a curve object to tweak its influence precisely based on the amount of input variables as well as scaling variables, if the input variables (e.g. the speed) should change their value range during development.

5.1. Trail renderer

Left: Both cavemen flying in the direction of the velocity, leaving no trail. Right: Black caveman is turning, rotating against its velocity, leaving a big black trail (which is also long due to high speed)

In the first flying prototype we already added a trail renderer to get a feeling for the movement of the cavemen. For our next step, we changed the shape of the trail depending on flight parameters, which created better feedback for the player as well as a really neat visual effect. The first major change was to the rotation. A trail only is created if the caveman rotates against his velocity. Since the caveman automatically turns towards his velocity if there is no input, the trail would serve as an easy indication which character is being controlled - an effect of utmost importance in a game, where two similar characters are controlled at the same time. This effect can be understood as “wind lines” that appear if e.g. a fighter jet rotates against its velocity.

5.2. Look ahead

Camera look-ahead serves two purposes: On one hand, it creates the impression of speed, looking like “everything is moving so fast that the cavemen cannot keep up”. On the other hand, it shows the player more necessary information: the direction they are going in. Also it shows less of the direction which they are coming from, which is unnecessary information.
The effect is easily implemented by moving the camera into the direction of the average velocity of both cavemen and give some parameters to change the influence. This is important, since the look-ahead should not be too large in the x-direction, as most screens are shorter in the x-direction and the characters should not leave the screen. Also we need to apply some lerp-smoothing, since characters can instantly change their velocity by pulling each other up, which would cause some unwanted jumps.

5.3. Parallax

The gameplay mechanics in Caviators are rather slow, to give the player enough time to make choices, and to provide reliable physics. To still create the fast, arcade-like experience we are aiming for, we tried adding a parallax effect.

This effect creates a problem, since a cave wall usually is pretty “2D”, so it makes relative little sense to have an object move slower or faster because of its z-position. While this is a small problem if there are only the characters on screen, it becomes disturbing if there are also birds on screen: If the characters drop, the cave wall moves even faster upwards, which makes it look like the birds are also dropping. To tackle this problem, we unbound the parallax in x-direction from the y-direction, so we could slow it down on the y-direction, while keeping the speed on the x-direction. This is no perfect solution, since it removes some feeling of speed when falling, but it is suitable for now.

5.4. Clouds

Cloud background (left), cloud foreground (right)

Red caveman is partly occluded, net blends at transition point, red squares are test birds

In the sky there are not many objects to get points of reference, which in turn leads to less feeling of speed. We decided to add clouds as objects, since they would not change our current gameplay as well as present an easy way to give a point of reference. Also, they look cool.
The clouds are created with two layers. While the background shows the complete cloud, there is a half cloud as foreground, which is placed in front of the cavemen. This way, they get partly occluded after half of the cloud, which gives the impression of 3D.

So far, clouds are easily spawned around the player and destroyed if the are too far away from the player. Later we could link them to the player height to indicate this gameplay value.

5.5. Wiggle animation

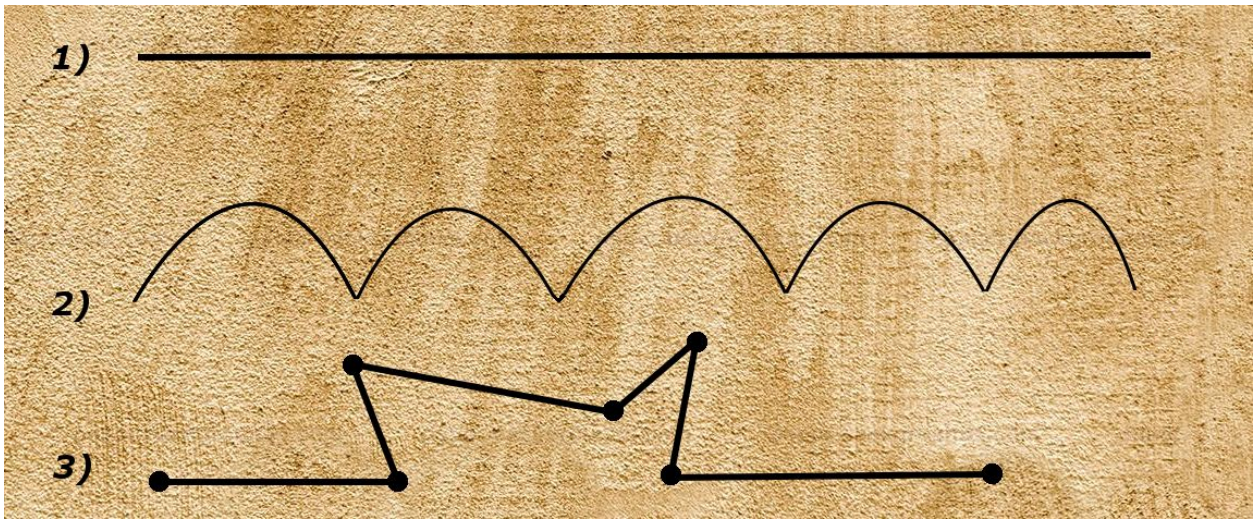
For our caveman model we currently only use one animation: Wiggling around. This was perfect to give some feedback on the speed: The faster the cavemen fly, the less they would move, since moving is hard under air pressure. Actually, they should also stretch their bodies. But this would require us to smoothly blend their animation to a specific point after they crossed a specific threshold - which is just too much for now. Also slowing down the animation proved to be good enough so far. The slower the cavemen are, the more they wiggle around, which gives a cartoonish effect if you rise up, thereby lose speed until you are “standing” in the air, wiggling strongly, and then dropping again. And it is good to remind you, that you are actually a caveman and you do not belong into the skies.

5.6. Screenshake

Screenshake is activated as the cavemen become extremely fast. It is important to start the screenshake only at really high velocities, since it will easily cause nausea if activated constantly. Also here it is crucial to deactivate the effect, since some players will not even tolerate a mild occurrence of the effect.

6. Bird prototypes

Since the behaviour of the birds has a direct impact on the behaviour of the player we have to offer several types of birds for a versatile experience.



The three different base movement types currently implemented.

6.1. Normal bird

Flies in a straight horizontal line (as seen at“1)”) with constant velocity and will appear alone or in groups of 3, 5 or 7. This is the “filler” bird, with a quantity over quality orientation. When hit by the net the bird will get knocked out and stop moving and will behave like an lifeless object.

6.2. Jumpy bird

A variant of the normal bird which will try to flee when the player approaches. Typically it tries to evade vertically or increases its speed for a short time.

6.3. Fat bird

This big bird defies all physical rules by moving slow and steady while still being airborne. When “caught” with the net without any preparation, it won’t budge and will probably bring an end to the current bird hunt. It may drag the players along with it for a while first.
To beat this bird, the player needs to smack it with a big enough mass, which leads to a knockout. After taking the hit, the fat bird will be knocked in the appropriate direction and float in the air until hit by the net, which will result in an again “lifeless” bird.

6.4. The long wing

With its abnormal long wings, this bird struggles to move steadily through the air (“2”.) It is the first to have a physics-based movement with an actual mechanic to “defy” gravity. The wave-like path is created by a initial horizontal velocity paired with a periodically applied upwards-force. The catch-difficulty and thus the reward of this bird can be varied with its “size” (this is just visually), since longer wings result in a larger upwards-force and a longer airtime. This way, only one bird type allows varied spawning options with different settings.

6.5. Hummingbird

Like its real life counterpart this bird moves fast and unpredictably with short stationary phases (as seen at“3)”). Also it only makes sense to spawn it relatively close to the ground or some trees. Its small size combined with the fast movement and its spawn location makes this bird really hard to catch.