



Interim Report: Code Bread

4th of June 2020

Team *Callstack Overflow*₂

Maximilian Werhahn

Mark Pilgram

Min-Shan Luong

Felix Neumeyer

Task Progress

Level Design

As of this milestone, the basic layout and wall structure of the space ship the game takes places on has been completed. This leaves filling the rooms with props and setting up lighting as optional steps to be completed at a later time.

For the most part this was created with the help of a low-poly sci-fi asset pack, as modeling this many varied assets of sufficient quality would have taken too much time. The layout of the ship is entirely original though, with some effort put into creating rooms that aren't necessarily rectangular with a fixed side length.

The layout contains several rooms connected via doors that for the most part form one big circle. Players can interact with doors to either lock them in an open or closed state, or just let them open and close automatically. This can be used to influence the environment simulation around the ship.



Our inventory system works pretty much exactly as it did in our prototype: Players can store one large item and up to either 4 or 5 small items, depending on whether the large item slot is empty. Large items are visibly held in the player's hands as they walk around.



Plant System

Currently our game has four types of plants. Three of them can be grown with seeds in plant pots, namely Sunflowers, Wheat and Tomatoes. Sunflowers will later generate an increased amount of oxygen, whereas Wheat and Tomatoes are pizza ingredient-generating plants. Plants require water and time to grow, if their water reservoir reaches zero, growth is paused. After their initial growth phase, ingredient-generating plants switch to a resource production phase which is shorter than the initial growth phase. All of these variables are tweakable for balancing purposes.

As plant pots are just a special type of pickup, they can be carried around in the player's inventory as large items while simultaneously growing plants.

The other type of plant are Mushrooms. A Mushroom is somewhat periodically spawned at a random spawn point in a random room on the spaceship, meaning that players will need to look for them to gather them.



Pizza Baking

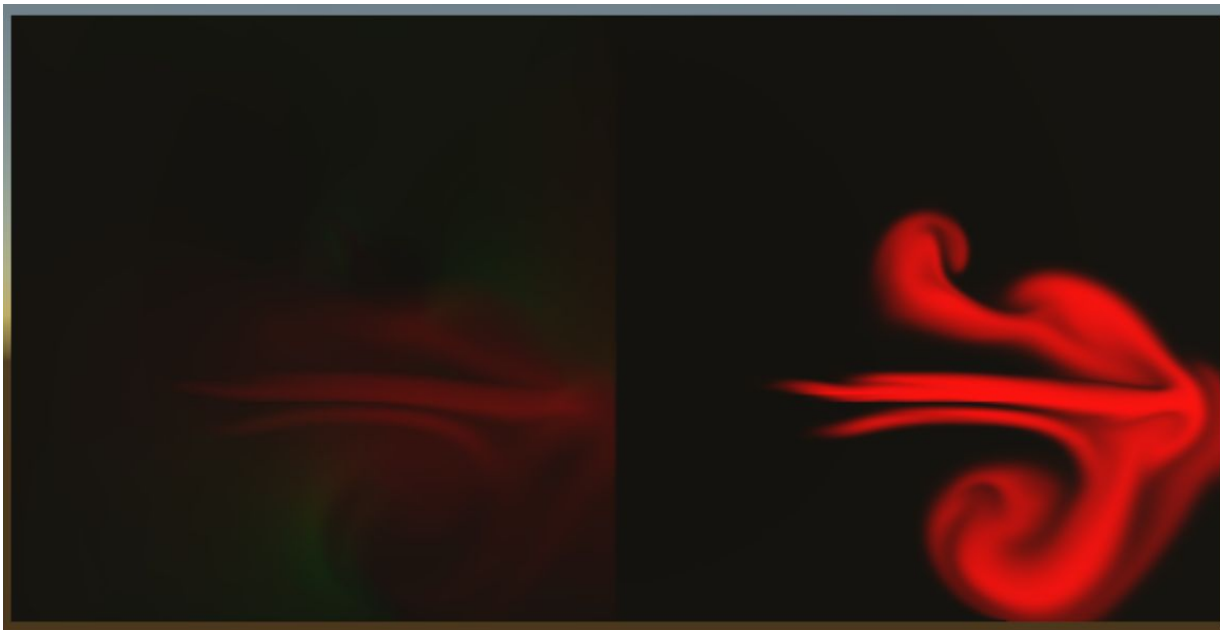
When interacting with the oven, players first get to choose the type of pizza they want to make. In this initial state, the oven will display the ingredients required to make each type of pizza. Once the baking process is started, the oven menu will switch to display the baking progress. Pizzas have individual minimum and maximum baking times. If the maximum baking time is exceeded before the pizza is removed from the oven, it will be transformed into burnt pizza.



Fluid simulation

We simulate the spreading of oxygen and carbon dioxide via an eulerian fluid simulation, parallelized with compute shaders (GPU). O₂ and CO₂ are modelled as a quantity and are stored at the central positions of a uniform grid. The velocity of the fluid, however, are stored in a MAC grid at staggered positions of the cells. To advance the simulation, first the quantities and then the velocity grids are advected. Afterwards the pressure solving step occurs, keeping the velocity divergence-free. As an advection scheme the Semi-lagrange and MacCormack approaches are implemented.

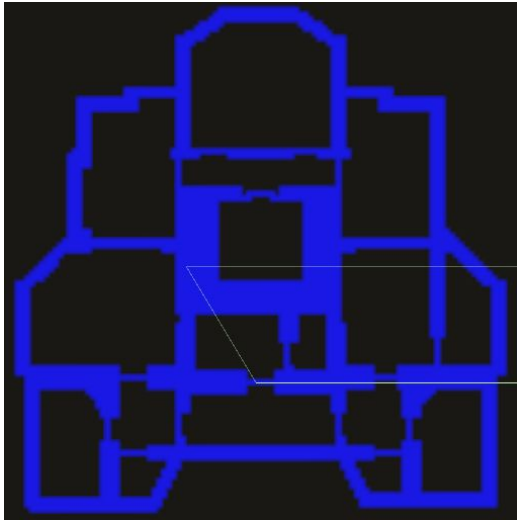
Additionally, to preserve more energy inside the fluid we also implement the Advection-Reflection scheme instead of the usual Advection-Projection. To solve the tridiagonal matrix system we use a fixed amount of conjugate gradient iterations (namely 50), because synchronizing between the GPU and CPU is rather slow and actually yields worse performance. However, to read or write to the GPU a fixed tickrate is used. During the synchronization process, there are several grid properties being updated, e.g. when a door opens the flag grid also updates at the corresponding position or if an object produces or consumes a quantity a inflow grid is updated, which is applied before the advection step.



Instead of just relying on the force/buoyancy, we increase the divergence of the fluid by adding the density of O₂/CO₂ scaled by a factor of 0.001. This way the fluid automatically spreads/diffuses in rooms even when the magnitude of the velocity is rather low. It might not look as realistic, however, it is easier to balance the consumption/provision. We also assume for our game that O₂ and CO₂ are our only type of fluid, and naturally it should

spread when releasing the quantities in a vacuum. The divergence control operation also enables the possibility of ship hull breaches pushing the oxygen into space.

Example of the underlying flag grid (showing solid cells):



When considering the advection/reflection scheme, a grid of the size 256x256, 50 CG iterations and a fluid tickrate of 1 second, the simulation runs at ca. 320 fps. Decreasing the tickrate to 0.1 results in ~300fps. Halving the grid size in each dimension, increases the fps to 380, mainly because less memory has to be transferred during the synchronization steps and the advection/pressure solving steps are cheaper. The hardware used includes a GTX 1080 & i7-8700k.

Fluid Visualization



We want the player to be aware of the fluid simulation that is driving the game. In the example renderings above you can see the lack of oxygen rendered in red. In the middle image you can see the oxygen floating into a room after opening the door to a oxygen-filled room. On the right image you can see an issue with this method: walls do not hold any oxygen and are therefore rendered with the red effect. For this effect we perform a "post-processing" pass, that projects the rendered pixels back into the scene using the

depth buffer. We then project the world position into the local coordinate system of the fluid simulation's texture to get oxygen/carbon dioxide concentrations at the respective pixel/world position for coloring.

System Design

We wanted to create a systemic game, meaning that the game has different systems that are linked to each other in a fairly generic way. Each system has inputs and outputs that can be connected if they are matching. This approach is often found in open-world RPGs. The most prominent example would probably be the Assassin's Creed franchise or Zelda: Breath of the Wild. In Zelda for example, the ground gets slippery when it rains, all metallic objects can conduct electricity and fire keeps you warm in a cold environment. This allows a game to be more convincible, when all systems can interact with each other. It also allows for emergent gameplay that doesn't have to be specifically coded for.

In the end we are only having few systems in our game that are somewhat connected. These are described in more detail below.

Life System

Living beings are all part of the life system, including plants. They all have a health component that can decrease under certain circumstances. E.g. through oxygen deprivation. If a player faints under these circumstances, she or he can be revived by other players.

Air System

Through the fluid simulation, we are simulating oxygen and carbon dioxide concentrations and diffusion in the air of the spaceship. Living beings, like the player and potentially animals at a later point in development, breath oxygen and exhale carbon dioxide. Plants on the other hand take carbon dioxide out of the simulation and turn it into oxygen.

If there is not enough oxygen at the position of the living being, it will faint (or die in case of the animals, probably) after a short amount of time (see above 'Life System').

Fire System



Fire can spread onto things and living beings in its surrounding which damages them. It is stylized and visualized using a shader, and a post-processing bloom effect. The shader basically slides selected noise textures to cause a flickering effect. The reason why we used a shader instead of a particle system is to avoid having too many



particles in the scene which may lower the performance (already used for e.g. space environment and explosion). Fire will at a later point in time also consume oxygen and produce carbon dioxide over time.

Additionally, we use this shader to only visualize the fire of the turbines.

Space Environment



In order to make our environment look more realistic, we use a particle system (specifically trails) that gives the illusion of being on a flying spaceship. These particle trails spawn randomly within a 3D box and move towards the bottom of the screen parallelly to the spaceship.

Currently, there are around 100 particles at once on the screen which does not noticeably affect the performance.


Challenges and non-Challenges

Fluid simulation

Implementing this system with real-time efficiency is rather difficult. Techniques such as groupshared memory are necessary to use, e.g. for the dot product computation which was implemented via a parallelized reduction. Because this operation is fundamental for the conjugate gradient algorithm, its performance is vital for the overall performance. Furthermore, the typical stopping criteria of the CG algorithm can not be applied, because it would lead to synchronization between the GPU and CPU each iteration. Instead it is faster to use a fixed amount of iterations (e.g. 50).

Additionally, extra-care is necessary when handling many random write resources with compute shaders - only a maximum number of 8 of such buffers per GPU kernel is allowed, which was a problem, because we accidentally specified a too large amount. This lead to the matrix system only being diagonal - the off diagonals were always set to 0. Therefore, the fluid behaviour was completely wrong.

System Design



While a systemic game design is great for various reasons, the additional complexity can be a hindrance when you want to create a game quickly. Since our game will probably have a relatively small amount of elements that could interact with each other, it probably doesn't make sense to code everything with the systemic game design in mind. While we have some systems like the Air and Life Systems interacting with each other, we did not go farther than that because it was slowing down the overall progress.

Universal Render Pipeline

While the Universal Render Pipeline (URP) is the way to go according to Unity, the documentation on all of its features is somewhat lacking at this point. There are some official samples on how to use it, but it's kind of hard to get started writing shaders and custom renderer features as the resources provided by Unity are not really guiding you through all the components involved.

Shaders need to be done slightly different and you can't use Surface Shaders for example, as they were a feature of the old rendering engine.

We also have problems with multiple cameras in combination with the URP. For some reason the depth buffers are not correct

Fluid Simulation Visualisation

Since we are using two cameras that introduced problems to the writing of our shaders that were somewhat unexpected. We had to rewrite some of the shaders twice, or change the way we add the shader to the rendering pipeline in order to make it work for multiple cameras. This was mainly because we were not able to render depth buffers to separate Render Textures, even though Unity provides us with a function that was supposed to do exactly that.

Handling Interactions from Multiple Players

Coming from developing single player games, having more than one player did mean that some aspects of the game needed to be approached slightly differently. Basically rather than being able to access single instances of scripts via a static reference, every interaction needs to be given the context of the player that's performing them. Overall though this wasn't too much of a challenge, and other than the way input devices are handled, expanding the game to include another player could be done in under a minute.

Our game is even capable of handling multiple players simultaneously interacting with a menu tied to a single object. If for example two players try to operate the door controls of a single door, both can independently select between its three different modes (auto, closed, open). Once a selection is made, the door's current state is updated on all opened menus, as the door keeps track of all active menus that are associated with it and can update information as needed.

Design Revisions

Burnt Pizza

Rather than allowing players to instantly craft pizza out of the required ingredients, pizza now takes time to bake. If the player doesn't pay attention and leaves the pizza in the oven for too long, the Pizza will get burnt.

