



HERBERUS

Technical Review



Sommaire

1. Introduction	p. 04
2. Characters	p. 08
3. Systems	p. 12
4. AI	p. 16
5. Tools	p. 18

Introduction



Herberus offered several technical challenges to overcome. According to the project needs, we decided to focus the code architecture on three key factors : flexibility, production efficiency, and performance.

The Herbecrew



William PATRUNO

Gameplay Programmer
Tool Programmer

william.pat@free.fr
patrunowilliam.com



Thomas DUBRULLE

Gameplay Developer
AI Developer

thomasdubrulle52@hotmail.fr

And many thanks to Yoan Garnier, our external AI Programmer

Introduction

First, the production was rhythmized by regular playtests, and therefore we had to be able to quickly change how things work from one playtest to another with as less hassle as possible. As a consequence a flexible global architecture was needed, to ensure that everything that needed to be tested could be checked on time.

Also, production efficiency is a key element on the project, and more especially regarding the level design. To increase the team's overall productivity, we created several tools to speed up the workflow -from gameplay element integrations to world building- and reduce the risks of mistakes while creating the game.




Last but not least, a lot of elements are displayed on the screen, and most of them gives some interactivity with the players. During several steps of the project, we optimized the code as well as rendering to provide a smooth experience to the players.

This document explains why and how we have organized the technical aspects of the project around these three pivots. It also contains clues about some of the difficulties and which are specific to this kind of project, and that we faced off,.



Characters



The four explorers of Herberus have similar abilities, but they are differentiated by different colours, textures and names. Moreover, their equipment and skills evolve separately throughout the game. However, in the code they are only differentiated by an identifier. Indeed, each player specific trait is held in a separate game object and distributed among all the players, so that the personal attributes are centralized, and to avoid copying each modification on each player.

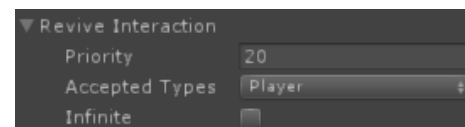
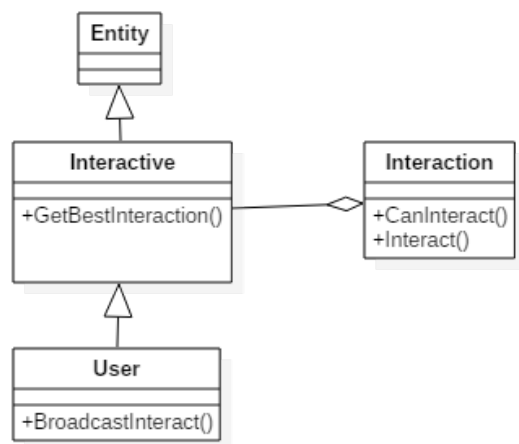
Interactions

The cooperation involves a lot of interactions between players but also with the environment. To answer this need, we developed a simple system in which you can add interactions dynamically. The code to make this happens is made of two classes : Users and Interactives. An User is an Interactive who can interact with other interactives (He is the one who uses the interaction). Interactions have priorities, so users will always use the best interaction available. For instance, you will always revive a friend before picking up an item.

Interactive elements

Our environment contains multiple interactive elements. Some of them interact with the player's items, but others like the battering ram require multiple players to work.

In order to do this, the MultiAgentControllable script makes a movement depending on the input from the players carrying it. To make a player locking on it and move it, we created a script called AgentLockPoint that is responsible of one player. When a player interacts with the AgentLockPoint, his inputs are transferred to the MultiAgentControllable, which then use it according to its type (controlled by two or four players, special group actions...).



Items

To further develop these interactions, we made a polyvalent architecture, which acts as the skeleton of the player-to-world and world-to-player interactions. It is based on two main scripts : Actions and Effects. The Item class is the base of all the weapons but change it's behaviour by adding different actions and effects.

Forcefield

Item

Projectile Action

Slow Effect

Medigun

Item

Stream Action

Heal Effect

Rifle Gun

Item

Hitscan Action

Damage Effect

Actions

To centralize all kind of actions under a general structure, we created Actions objects. These objects regroup how an Entity in the game do something. For instance, an action can symbolize a fired projectile, a stream between two entities, or an explosion. Every Actions have the same basic behaviours, like a cooldown, the known targetable entities or other additional conditions specific to each Action. Also, they use Effects to concretely interact with game entities. When an action is launched, it is autonomous and trigger their Effects on entities according to the kind of Action.

Effects

Effects are what the action do when it succeeds. It ranges from damaging an enemy to pushing a physical object in the world, and can even trigger environmental events. All effects are launched when an action is being or has been achieved. Effects are the atomic sources of interaction between entities of the game. Since a single effect focuses on doing one specific thing, the system can be used to add several different effects on a single action and swap them as the project is evolving. Moreover, the structure is lightweight, and effects can be quickly coded. As a consequence, it allowed us to prototype, add and check each feature with very small time costs.

Systems

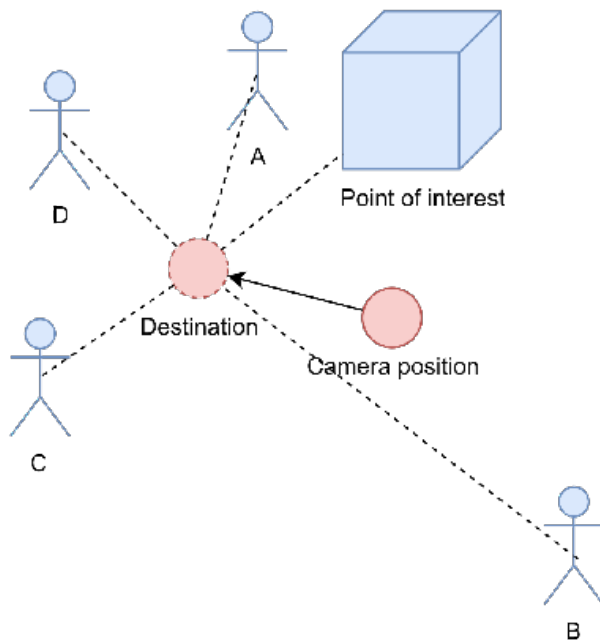


One of the challenges during the production involved the camera. Indeed, all characters need to stay within the screen view at all time, so that the players do not lose sight of their avatar while playing. Also, the team wanted to include some camera scripting, in order to focus on different elements of the game and create points of interest in the world.

Camera

In order to do that, the barycenter (or averaged positions) of all players and important game elements were used to know where the camera should be going. We then converted the camera's view into world coordinates, and deduce if any player will go outside the screen if he or the camera moves to its intended direction, and calculate the best movement to keep them in the limits. Finally, to further extends the possibilities of the camera, several parameters were added to change its position, rotation or distance to the players.

The main difficulty in the process lies in the difference of the shape of the limits : Screen coordinates are limited inside a rectangle, but due to the perspective, it is transformed into a trapezium. Even though it involves more calculations (mostly in geometry), it allows to fine tune the movements at the edges of the camera, and to get the behaviour we wanted.



Stele

Steles are artifacts from an ancient civilization which have the power to improve the technology of the explorers. A stele can give only one upgrade per player. So they have to choose carefully which upgrade they want depending on their own play style. All steles contain four modules : one for each member of the team.

Module

Modules are objects which contain upgrades. they have to be activated by a generator before it can be used. In order to link generator to a module we use an event system.

Upgrades

As our players upgrade their characters at the end of each level, their weapons have to be modified. Thanks to our flexible architecture, upgrading a weapon is easily done, since we only need to swap effects scripts between each other. Moreover our effects and actions script are divided in two scripts. Also, each of the two classes have one functional part -the one with all the logic- and one dataset which contains all the variables, like game variables or visuals FXs. When a weapon is to be upgraded, the dataset is generally the only one that needs to be changed. It is especially useful when we want to save our weapons as we only have to serialize the data, and not all the weapons and subtypes.



Trigger Event System

Our game designers needed to trigger some events for the music or to spawn enemies depending on where the players are or what they are doing. To help them, we made a trigger event system which do not need any programming skill to use, so that they can use them freely without a programmer at their side.

To use this system, the team only have to add conditions to a script and add events when conditions are met.

A specific case : the Multiplayer UI

Because we want to push the cooperation as far as possible, we didn't want to give all the controls in the menu to the first player or to let everyone decide randomly by just pushing buttons.

Therefore, we made a main menu as a small level with some elevators which leads to the different levels or to quit the game. All players have to be on the elevator in order to make it move. So they have to organize themselves even in the main menu in order to play. To achieve this, we used the trigger event system to make sure all players alive are on the elevator.

Then we made a multi user button system for the pause menu where you have to hold down a button in order to validate a choice but if another player push another button, it cancel the filling. While there is more than one button down, no buttons fills.



AI



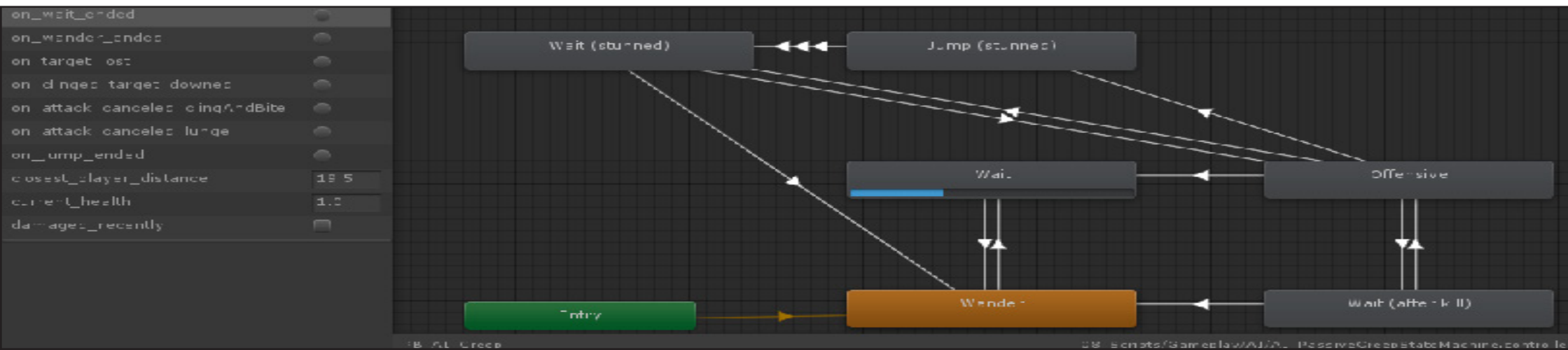
Behaviours

Along the production, the need to quickly make and check different behaviours arose. In order to do that with little production cost, an AI state machine was created. This system is set into the Unity animator controller, so that the current behaviour can be easily visualized and tested.

To bind the AI behaviours to the body it controls, we introduced a generic system that can perform general orders -like moving to a destination and taking damage-, as well as performing others that are specific to the AI type : Jumping on a player and damage it, play an animation once, firing from a distance and so on. With this architecture, the AI can have specific behaviours with special gameplay effects, while retaining the general utilities and features AIs have.

Interactions with players

To make the different AIs interact with the player, we have reused the same Action/Effect system as the player. This provides us with several advantages : code can be factorized -since the behaviour is globally the same-, which in turn lowers the probabilities of technical difficulties we may encounter during the production, as well as increasing the productivity.



TOOLS

The background image is a 3D rendered scene from a game. It features a landscape with dark, reddish-brown rocks and sandy ground. There are several purple, spiky plants and some orange, star-shaped flowers. A small, dark-colored character with a yellow backpack is standing on a patch of sand, surrounded by a green circular highlight. To the right, there's a large, blue, segmented structure that looks like a giant's leg or a piece of machinery. In the bottom left, there's a small, white, cylindrical object with a purple top and a white symbol. The overall lighting is warm, suggesting a sunset or sunrise.

In order to speed up the production and to answer some of our needs we made some tools on Unity.

Player Input

As we have four players we needed to take input from four controllers. the default input system of Unity did not answer our needs because it was not easy to configure and it also cannot control joystick vibration.

These are the reasons we have decided to make our own input manager, which is designed to access easily multiple controllers and integrate additional functionalities. Thanks to this, an unique identifier bound to the player can access the controller that is handling it. Additionally, the button mapping can be configured directly on the player by a single click, which has been useful to improve the control mapping during the production.


Level Design

As Herberus is a planet with a lush and rich environment, we made a paint tool in order to help us to create levels. The paint tool allow us to easily place a lot of assets in the scene and make sure objects are spawned correctly on the ground while granting others parameters, like a random size, orientation or specific spread.

Optimization

Game performances were a recurring problem during our production. Indeed, some assets are numerous and/or costly, and, as they were added into the game, the framerate suffered from several drops which affected the players' comfort. These elements included the multitude of visual elements and more specifically animated assets.

To solve these issues, we included a set of optimizing tools and worked our code as much as possible on triggers and events so that most calculations are avoided each time the game is refreshed. For instance, for bushes that react to player movements, we reduced physics calculations by reducing them to only the colliders we need to move with, and deactivate them if the characters are too far from the plant and, as a consequence, will not interact with it.



Finally by checking where the bottlenecks are (by using the Unity profiler for example), we were able to only work on what actually brings down the framerate. This significantly reduced the time to improve calculations without compromising the production of other elements.