|  |
| --- |
| Continuing Education Program |
| Prototyping in Unity |

**1- What is a prototype?**

|  |
| --- |
| **Copyright Notice**  Copyright © 2013 DigiPen (USA) Corp. and its owners. All rights reserved.  No parts of this publication may be copied or distributed, transmitted, transcribed, stored in a retrieval system, or translated into any human or computer language without the express written permission of DigiPen (USA) Corp., 9931 Willows Road NE, Redmond, WA 98052  **Trademarks**  DigiPen® is a registered trademark of DigiPen (USA) Corp.  All other product names mentioned in this booklet are trademarks or registered trademarks of their respective companies and are hereby acknowledged. |
|  |

In game development, a prototype is a proof-of-concept of one or several features of a game. Prototypes are done to show the feasibility of a design, and are not needed when making a game in a very established genre. However, since most games have at least one system that is somewhat unique to them in how it relates to other systems, a prototype is almost always done.

For real time twitch-based games, prototypes take on another function. Real time controls and behavior is hard to get right, and since it´s the lowest level mechanic of such a game, it needs to be perfected before any proper structural design can be made.

As an example, think about Super Mario Bros. In order to design enemies and levels for that game, there are some constants that have to be known, specifically speed, jump height and jump length. **Design in such a game CANNOT progress unless the mechanics are FINAL**.

As such, prototyping basic input mechanics has a more strict set of requirements than more high level mechanics. The objective of the prototype is not to prove the mechanics can be done, but that they work, that they feel responsive enough and that they **allow for structural variety**.

As such, building a prototype has three basic steps, these steps are started in the following order, although it´s not necessary for one to be fully finished before starting the next:

1. Coding the basic functionality. This step is normally the easier, especially if the team is using an engine they already know. If the engine is new, they will spend time acclimating to it.
2. Tweaking the basic functionality until the response is as good as it can get. This sometimes requires recoding (if some functionality can´t be tweaked to perfection) but mostly it is about tweaking variables. This is where bugs with the main mechanics need to be addressed, since there´s no guarantee those bugs could be resolved further down the line without affecting core mechanics.
3. Providing context for the mechanics, and ensuring they allow for an “expressive range” that is enough to support the desired scope of the game. The longer and more complex the original vision is, the more interesting interactions between mechanics that need to be devised. We need to make sure the mechanics are complex enough to allow for variety of level design and mastery of the mechanics. If the mechanics are too simple so we can´t provide significantly different challenges, it might be a symptom that we might be missing something (normally extra mechanics that interrelate with the ones already present).

Once the prototype is finished, even if the mechanics need to be coded as to integrate with the broader system architecture, it´s values should not change. Game development being iterative, this is not always the case, but changes in core mechanics once the game has left prototype stage can be problematic.

**2- Grey boxing and gameplay without art:**

It is perfectly possible to prototype without art, although that´s not always the case. Big studios and teams need to prototype their art as they prototype the gameplay. Game art has certain technical requirements that are tied to the core code structure.

A prototype without art is called a “grey box”. The name comes from the grey look of simple meshes without textures. The advantages of greyboxing carry over from the prototype stage into proper production.

This is because if a game is greyboxed with specific set of geometry (for example, prototyped in an environment in which the smallest piece of scenery has half the size of the main character), the features of that geometry need to be maintained throughout the whole game.

Therefore, to avoid level designers and artist to include features that are not compatible with the mechanics, levels are designed first as grey boxes.

This is because grey boxes represent the logical dimension of the game. The wall of a grey box is a wall for the logic of the game. When the final art goes into the game, even if that art does not follow precisely the grey box, the game logic will.

This allows to **separate art from gameplay logic**, and is a fundamental part on structuring a game project.

There are problems that can arise from the use of grey boxes, most notably object clipping and invisible walls. Those problems can be solved by separating the visual representation from the gameplay one, and keeping logic intact.

For example, consider the following: The blue ball is the player and the blue square is the collider for gameplay logic. The red square is the visual representation of the wall (it´s a 3D space and it does not follow precisely).

If the wall hits the wall, clipping would occur (on the right the wall obscures the ball):

To solve this we could add a collider to the visual representation of the wall, ray cast against it and move the visual representation of the ball:

This might seem counterintuitive and expensive (this kind of solution tends to be found in higher end games) but it solved the visual consistency problem without affecting the logic behavior.

The alternative would be to live with the clipping, since using the visual mesh for collisions (unless the prototype has been created to support such fine grained behavior, will in most cases break gameplay.

This greybox approach allows several team members to work in interrelated features of the game without creating gameplay conflicts, and is to be encouraged.

**3- Game Feel. When a prototype works:**

So how do we judge a prototype, besides knowing that it “just feel good”?

This is actually a very complicated issue mostly related to game design. There is no right answer, but a series of guidelines we can follow. As a rule of thumb, for real time twitch based games, a prototype has to:

-Be responsive. Have good controls.

-Provide a satisfying simulation of the desired metaphor.

-Have the right level of polish.

Let´s examine each of these one by one:

**1-CONTROLS AND RESPONSIVENESS**

Probably the most important thing for a game is to **show a response in real time to every player** input.

By response here we are encompassing every kind of visual, aural and tactile feedback, but most of the time this response should imply a **perceptible change in the simulation**.

This means the change cannot be too slow or too small. What constitutes a too small change will depend on the player´s perception abilities, and needs to be tested, but as a rule of thumb we are talking about a significant change of color, shape or movement. It is always wise to be on the side of exaggerating this kind of feedback to input, rather than risking it go unnoticed.

However, what we can do is provide some metrics in terms of response of speed. A regular human being needs 10ms to **acknowledge** something has happened. That means tha before 10ms, there is a chance the effect has not even being processed by the player. This means every response to input, be it feedback or simulation change state, should last for more than 10ms. Otherwise the player will perceive something but not have time to process it, which will create a confusing feeling.

Also, a player takes around 5ms to process a response send his response to the input. That means that **the quickest we can expect a player to react is 15ms**. Reaction times quicker than this border the unplayable.

But more meaningful for a game feel is that after those 15ms have passed, the player needs another 10ms to apprehend the new state of the game. That means that there´s a window between 15 and 25ms in which a response to an input will be perceived as real time. More than 25ms will feel sluggish and more and more broken the longest it takes, while less than 15ms can feel abrupt and jerky.

This mean the best implemented inputs do not trigger immediately. Input mappings have three sections, attack, hold and release. Consider the following curve:

This shows an input that slowly builds up over time, maintains a constant max level, and releases slowly over time. This kind of mapping is effective in every kind of game.

The previous curve is the horizontal movement curve from Super Mario Bros. while the next one is the curve from Mario Bros (the earlier game):

In practice this means the later game has an acceleration based movement (the speed is not immediately set to max, and inertia built into the system (if no input is present speed slowly decreases). The earlier game, following a more basic implementation, just mapped a constant speed to directional input.

Having acceleration allows the later game to not overreact to quickly and avoid jerkiness.

As basic as this is, for every input into a game system, correct values for attack and release are needed (and they don´t need to be linear, change can be mapped to a curve).

**2. SIMULATION**

Once the inputs are mapped, the reactions to those inputs (the “logic” of the mechanic in question) becomes the next important thing. Mechanics in real time games simulate realities. These realities can be as abstract or as concrete as the game requires, but the level of simulation must remain coherent.

As an example think about racing games.

The most realistic end of the spectrum, simulation games, have all kinds of physical simulations applied to the game object to precisely replicate the behavior of a real car.

However, a game like Mario Kart has completely different systems that create a simulation that is much more loosely based on anything real.

The level of visual polish of both games will have to be coherent with the type of simulation present. With the **metaphor**.

Also, please note that simulation precision is always related to gameplay mechanics and feel. Even in “realistic” racing games, like Gran Turismo, the physics of the car diverge from realistic behavior in many cases, most notable in car to car collisions. A real collision of two massive objects should bring those two objects to a stop, but racing games have “waterslide” physics. Collision between objects push them around but very seldom do they reduce the speed significantly.

This is because the main feel the designers usually want, no matter how realistic the metaphor, is that of speed. Realistic collisions will conflict with the feeling of speed, so things are tweaked to embrace the specific aesthetic the game needs.

As a rule of thumb, the simulation (the internal game logic) should be fine tuned for the specific feel we want as designers, in most cases going away from realistic physics and creating our own specific behavior for simulation situations (inputs, collisions, physics and anything that changes the game state).

Simulation includes things like hit points, life bars, rechargeable shields, collectible coins… the same simulation with different inputs and visuals would play the same, although it won´t necessarily “feel” the same.

**3. POLISH**

By polish we mean every single piece of feedback that is not directly related to the simulation. That is, any piece of feedback that has no repercussion on the game simulation.

Things like particles, sound and animation are part of the polish, and greatly affect the feel of a game. Even a sound cue can change the feeling of a collision, making something feel soft or hard, regardless of what we see.

Although prototypes and mechanics can be implemented with no polish (that is, they can be certified to work properly and being complex enough) it is important to have a cursory sense of polish already in the prototype, since that is going to influence the feel of the game (and thus how engaging it is). You could have implemented perfect Mario Kart mechanics, but if your polish suddenly becomes hyperrealistic, you will end up with an uneven product, no matter how precise the mechanics are.

Most triple A games prototype with almost final quality art, although with a small amount of it.

It is possible, though, to add a lot of polish without final art assets. Objects can be deformed and stretched without needing animations (although they would improve the end result) and functional particle effects are not that hard to do. There is no excuse for not implementing the right amount of polish in a prototype.

How game feel is impacted by different inputs, simulation and polish is best explored in the unity project that accompanies this document.

**4- Beyond the prototype. System architecture in Unity:**

A prototype should not be extended into a full game. One of the advantages of Unity is how easy and fast it is to iterate prototypes with it, but a common mistake that leads to complex project management is to use the prototype code into the full game by extending it.

It is important, once the scope of the project is defined, to do system design and to isolate functionality in an approachable manner. Unity´s component system, while very useful for interactive behavior and game logic, posits some problems when trying to adapt it to a more classical framework.

In a way, we need CONTROLLERS. That is, general classes that live at the top of the project and that provide the most basic functionality for inter-system operations. Three of the most basic controllers that are found in almost every Unity game are the INPUT CONTROLLER, the HUD CONTROLLER and the LEVEL CONTROLLER.

INPUT CONTROLLER: By isolating the “reads” our game logic does of the input state from the actual reading of the input devices we gain consistency of code (we have more control over inputs and their mappings) and portability of inputs (by rewriting the input class to allow for diferent devices we can have many different kinds of inputs without affecting game logic at all).

LEVEL CONTROLLER: Game stats, like score, current level, and even current health, are best isolated from the more direct player avatar behavior. This solves problems with player avatar destruction and provides the program with an extra level of security. If wel implemented, the LEVEL CONTROLLER can even solve the problems of saving and loading games. An ideal LEVEL CONTROLLER will have all the information necessary to load a game state (normally from the beginning of a level) and thus serialization of a game state boild down to serialization of the variables of the level controller.

HUD CONTROLLER: In big games, HUD coding tends to be the last thing to be finalized and polished due to its complexity and how prone it is to errors. To avoid this problem, we can isolate this code and manage it from a single master method. Although independent behaviors can call GUI code from the Unity API, the order in which GUI code is interpreted is in general very important, especially when dealing with menus. By having a HUD controller read our GAME CONTROLLER and provide appropriate feedback, we can have safer and less prone to bugs HUD code.

Isolating different systems allows for faster and safer iteration and polish of those individual systems without risking the stability of the whole project.

These systems, and any other high level controller, should always be placed as component of the Camera. That is because we need a single location that is coherent between levels to look for the controllers, and because every scene is guaranteed to have at least a camera.

For a simple implementation of an input manager, look at the code provided in the example project that comes with these notes.