OULUN AMMATTIKORKEAKOULU

Juha Keränen and Jouni Mäkelä

STRUCTURED GAME CODE

Development of a Game Foundation Library for the Modern Age

STRUCTURED GAME CODE

Development of a Game Foundation Library for the Modern Age

Juha Keränen and Jouni Mäkelä
Bachelor's thesis
Spring 2016
Business Information Systems
Oulu University of Applied Sciences

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietojenkäsittely, Web-ohjelmointi

Tekijät: Juha Keränen ja Jouni Mäkelä
Opinnäytetyön nimi: Structured Game Code
Työn ohjaaja: Teppo Räisänen
Työn valmistumislukukausi ja -vuosi: Kevät 2016 Sivumäärä: 30

Peliteollisuus on äärimmäisen kilpailullinen ala, jossa pitää pienin kustannuksin ja lyhyessä ajassa saada tehtyä toimiva ja viihdyttävä tuote loppukäyttäjälle. Useat valmiit pelimoottorit tarjoavat kyllä suuren määrän työkaluja, mutta eivät mahdollista työskentelyä matalalla tasolla, jolla voidaan helpommin varmistaa pelin korkea suorituskyky ja koodin eheys.

Hiillos-pelikomponenttikirjasto tarjoaa pelikehittäjille kokoelman työkaluja, jotka ohjaavat pitämään sekä pelimoottorin arkkitehtuurin että pelikoodin tiukasti organisoituneena, mutta silti avoimena välttämättömille muutoksille.

Kirjasto koostuu neljästä kokonaisuudesta: scene-manageroinnista, entity-manageroinnista, asset-manageroinnista, sekä kontrollista. Scenet ovat abstrakteja kokonaisuuksia, joissa pelilogiikka suoritetaan. Entityt ovat erilaisista ominaisuuksista koostuvia palasia, joista esimerkiksi pelien objektit muodostuvat. Assetit ovat pelin resursseja, kuten tekstuureita, fontteja ja ääniä. Kontrolliluokat saavat entityt tekemään asioita sceneissä.

Komponenttikirjastona Hiillos ei ota kantaa siihen, mitä muita kirjastoja pelin kehittäjän tulee käyttää kehittämisprosessissa. Mikäli kehittäjä haluaa käyttää Hiillosta omassa projektissaan, tarvitsee hän lisäksi renderöintikirjaston, sekä pelin tarpeista riippuen muita kirjastoja (esimerkiksi fysiikkakirjaston ja käyttöliittymäkirjaston).

Asiasanat: peliala, peliohjelmointi, peliteollisuus, ohjelmointi, c++

ABSTRACT

Oulu University of Applied Sciences
Business Information Systems, Web programming

Authors: Juha Keränen and Jouni Mäkelä.
Title of thesis: Structured Game Code
Supervisor: Teppo Räisänen
Term and year when the thesis was submitted: Spring 2016 Number of pages: 30

The game industry is an incredibly competitive field where you have to deliver an entertaining product quickly and cheaply to the end-user. Most of the preexisting game engines do offer a myriad of tools to the developers but don't enable working on a lower level where it is easier to ensure a high performance and keep the game code organized.

Hiillos game foundation library offers game developers a collection of tools, which help in keeping the game engine architecture and game code strictly organized, but still open for necessary adjustments.

The library is composed of four components: scene management, entity management, asset management and control. Scenes are abstract components where the game logic is executed. Entities are blocks composed of properties that can form e.g. the game objects. Assets are game resources, such as the textures, fonts, and sounds. Control classes establish entity behavior in a scene.

Hiillos game foundation library does not dictate which libraries the game developer must use in the game development process. If the developer desires to use Hiillos in a project, he also needs a rendering library as well as other libraries, depending on what the game requires (for an example, physics library and user interface library).

Keywords: video games, software, programming, game industry, c++

CONTENTS

1 INTRODUCTION

In a perfect world game programmers would be able to craft their games effortlessly out of modules that fill their needs perfectly. This way we could achieve binary size reduction, better memory footprint than heavyweight engines, avoid bloated libraries, achieve better performance, and have the engine be tailored for the game. In the spirit of UNIX philosophy the developer benefits when they have access to tools that work together and are crafted for a specific use. In this sense things would be modular which means that you'd only pay for what you use.

Unfortunately, we don't live in a perfect world. The tendency for complex, heavy, monolithic game engines, such as *Unity*, seems to be prevalent in the present-day game industry. This leads to a situation where all games at their core are just copies of each other not tailored for their intent.

As it is, we already have good libraries available (such as the *Simple and Fast Multimedia Library*) that manage specific low level tasks incredibly well. However, making a game is a demanding job and a lot of the functionality can be done for the developer even before they start working on their project.

The product of this thesis, *Hiillos Game Foundation Library*, helps with structuring the code, managing resources, and creating approachable but sophisticated artificial intelligence behaviors. *Scene management* isolates parts of the program code to their own independent modules which helps the developer to only work on specific segments of the game. All the game events occur in *scenes* and the *events* are driven by *entities* when they perform *actions*.

All games have entities, which in this context are game objects, such as player, enemies, props, and items. Entities have *properties* which define them. They can sometimes perform actions which might for an example move the entity around or make it perform stunts.

Asset management helps the developer to dynamically load *assets*, such as graphics and sounds, and effortlessly keep track of them. Hiillos also provides a support for *behavior trees* and a concept of *tasks*, which make defining the artificial intelligence easy for the entities that happen to require it.

We could build all the libraries required for a game ourselves, but there already exist a number of well-defined, well-tested, and well-stabilized libraries that are easy to use and have the support of massive communities behind them. Additionally, building libraries that, for instance, render graphics on a proficient level and on multiple platforms would be very time-consuming. There is no reason not to use the preexisting libraries as a foundation for a game engine.

2 THIRD PARTY LIBRARIES

The third party libraries used in the development of Hiillos are not parts of the library but rather tools that have been selected to test the foundation library with. The developer of a game engine has the freedom to use whatever libraries they wish to perform these tasks.

2.1 SFML

Simple and Fast Multimedia Library, or SFML, works as an interface between the engine and the operating system. It offers functionality for handling windows, audio, input, and rendering graphics with OpenGL.

SFML was selected for the test game engine over interfaces such as *SDL2* because of its native C++ implementation and encapsulated utilization of *OpenGL*. One important factor was also the cross-platform support. The Hiillos game foundation library is mainly developed on Linux platform but simultaneously the main desktop gamer demographic uses Windows as their operating system. For this reason cross-platform support was essential for the development of the engine while simultaneously allowing the users of Hiillos to develop games for a wider audience.

Without an interface like this the development of an engine would require a lot more work and intricate knowledge of how OpenGL works as well as how every operating system handles the creation of windows, playback of audio, and input management.

Alternatives: SDL2, GLFW

2.2 Box2D

Box2D is a 2D physics engine for games, but with Hiillos, the test game engine used it primarily to do the collision detection which would be extremely difficult to implement to the same extent without extensive testing and vast amount of resources.

There are other free, open-source physics engines out there, but the reputation and ten-year-long, still ongoing development of Box2D, during which it has been thoroughly tested, makes it the obvious choice as the physics engine of choice when working with Hiillos.

Alternatives: Chipmunk

2.3 ImGui

> *IMGUI IS DESIGNED TO ENABLE FAST ITERATION AND EMPOWER PROGRAMMERS TO CREATE CONTENT CREATION TOOLS AND VISUALIZATION/DEBUG TOOLS (AS OPPOSED TO UI FOR THE AVERAGE END-USER). IT FAVORS SIMPLICITY AND PRODUCTIVITY TOWARD THIS GOAL, AND THUS LACKS CERTAIN FEATURES NORMALLY FOUND IN MORE HIGH-LEVEL LIBRARIES. (CORNUT, 2016)*

ImGui is a graphical user interface library and was used with Hiillos to build menus, graphical controls, HUD elements, and debug tools for the test game engine. Unlike traditional graphics libraries such as *Qt* or *GTK+*, ImGui is not meant for an average end-user, but rather for 3D applications, embedded applications, games, or any applications where operating system features are non-standard.

As ImGui is very much bloat-free and designed to run as efficiently as possible, it is ideal for game development, where performance matters more than in most other fields of software development.

Alternatives: Nuklear

3 HIILLOS OVERVIEW

Hiillos by itself is not a game engine but rather a *game foundation library* that offers game developers a collection of tools of which they can choose the ones they find relevant to their game engine. Hiillos doesn't address the issue of rendering, user input, or sound playback. This means that the developer can use the libraries of their choosing for these tasks.

Hiillos is a game foundation library created with the intent to make the development of engine and the game tremendously more organized. When creating a game with Hiillos, the developer should follow the architectural design principles it introduces. Hiillos entities are very loosely coupled as they only have properties, which can be altered by actions. An entity should not know its key bindings nor how it's rendered in the game world. Scenes are logical containers which hold entities, as well as the game logic, in a restricted scope.

3.1 Scenes

Scenes are a tool for organizing the game into self-contained scopes. Games can get very large very quickly which calls for a method to divide them into pieces that are intelligible and practical. This helps the developer to maintain these pieces and make changes constrained to a specific scope of the game.

Initially, the game could be divided into a menu scene, game scene, and credits scene, but when the game starts to get bigger, the game scene might be further divided into smaller scenes (level 1, level 2, etc.) The job of a *scene director* is then to keep track of which scene is running, what comes next, and what data might need to be transferred over.

3.2 Entities

Entities are models of physical things (e.g. player, tree, light) in the game world and they are confined in scenes where they are created in. Entities are described with properties (e.g. hit points, jump height) which are universal to the game and can be used and altered by other entities and elements.

The behavior of an entity is modified with actions (e.g. attack, jump). This allows the developer to define functionality of the game independently and then simply bind it to any entities that happen to require it.

3.3 Assets

The way Hiillos handles assets uncouples them from the game code. When assets are needed they get loaded into an asset cache and get rendered when a scene calls for them. The assets aren't called directly because this would mean that the developer would have to know which assets they have available. Instead, the developer hands the *asset resolver tags* and the resolver then returns an asset that matches those tags as accurately as possible. This makes it possible, among other things, for the programmer to focus on the game code and the graphics artist to fill in the art assets as they get completed.

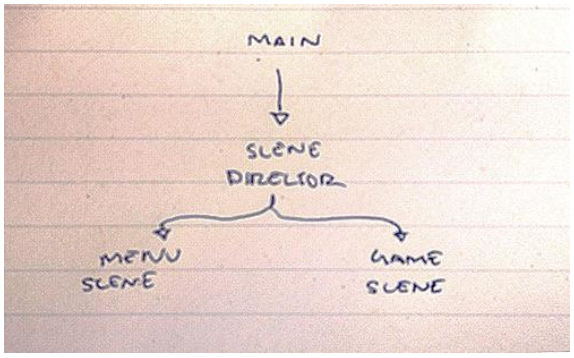
3.4 Control

The control classes are defined separately from the entities, who never know what controls they have. Key presses trigger commands which then in turn can be bound to the actions of an entity, such as the player.

Controlling A.I. is a bit different from controlling the player character. Hiillos provides a behavior tree functionality to the developer which allows them to assign tasks to the A.I. which then in turn perform actions based on their tasks.

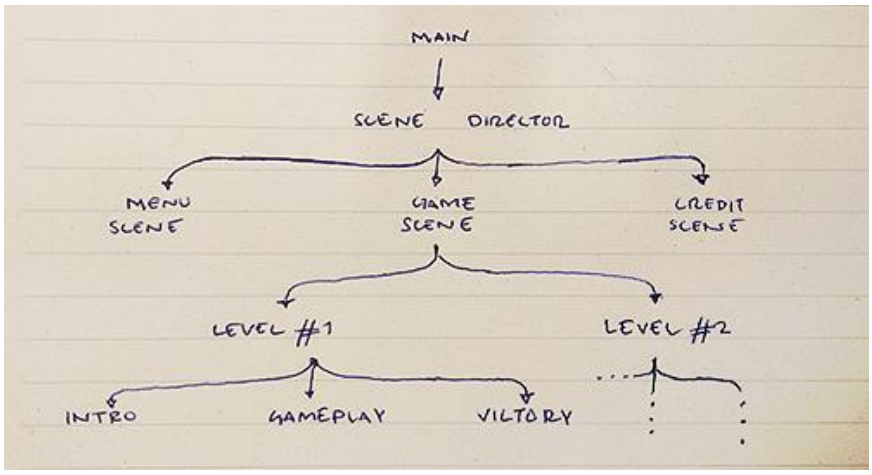
4 SCENE MANAGEMENT

Scenes are a tool for organizing the game into reasonably-sized segments. Scene itself is meant to contain a part of the program code that runs from beginning to completion (e.g. one game level) and a scene director controls the flow of the scenes and conveys messages between them.



PICTURE 1: Simple scene management

In its smallest form the scene management could contain one scene director and only a couple of scenes (see PICTURE 1) while a larger game could have multiple sub-directors controlled by a main scene director (see PICTURE 2). (Ewald, 2012)



PICTURE 2: Complex scene management

In the above picture the game scene is divided into two child scenes (level #1 and level #2) that have their own scene directors and child scenes.

4.1 Scene

When creating a scene the developer has to define two routines: update and render. The former should contain logic to update the status of the scene while the latter contains the logic for the rendering. It's important to note that Hiillos game foundation library doesn't do any rendering in and of itself but rather allows the developer to decide which library or method to use.

4.2 Scene Director

Scene director is a class that holds all the active scenes and knows how to create an instance of a new scene. The developer must register scenes to the scene director beforehand and can refer to a certain type of scene with a string.

By creating a scene director inside of another scene the developer can nest scenes and thus create more intricate scene diagrams. The main scene director is defined at the start of the application and it handles all the primary scenes and the derived scene directors

5 ENTITY MANAGEMENT

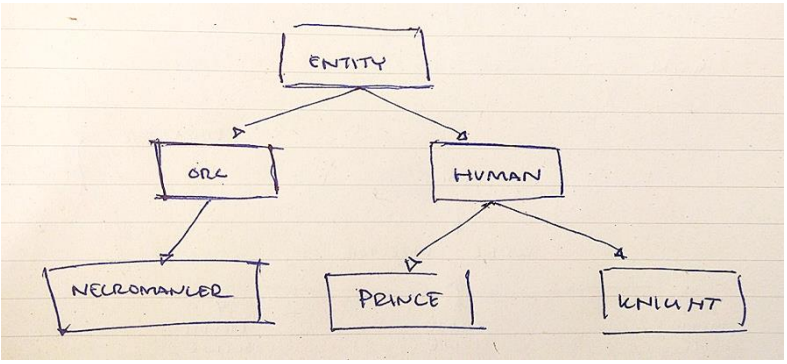
Most games have some sort of game objects. These might include the player, an obstacle, an enemy, and so on. Some objects might not even be visible but rather just affect the game world somehow, such as lights and the wind.

5.1 Entity

Game objects in Hiillos are handled with entities. Entity in and of itself is nothing but a base on which the developer starts to build a game object. This is achieved by defining properties and binding them to the entity.

5.2 Management Methods

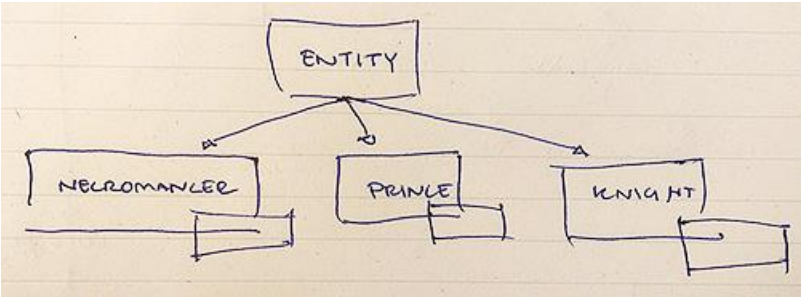
There are a lot of ways you can do an *entity system*. The traditional way of doing it is a *class structure* with inheritance.



PICTURE 3: Entity system as class structure

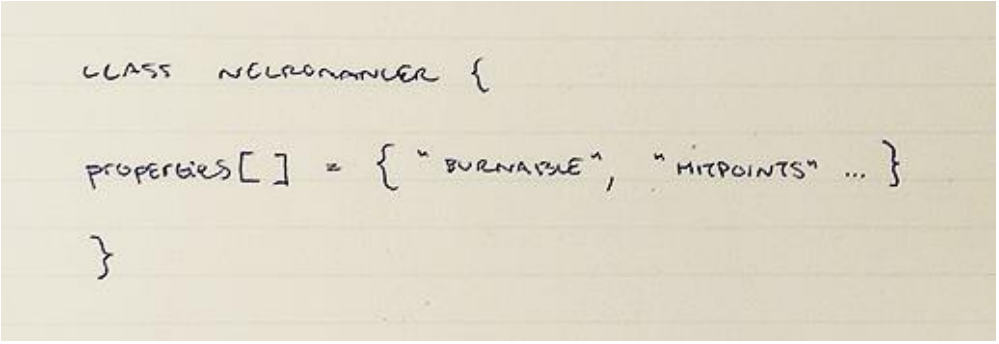
This is no longer considered a good idea because often in reality the properties of the *child entities* aren't restricted this heavily. Two child entities might require a similar property but don't share a *parent*. For instance, if you look at the example in PICTURE 3, prince and knight share the properties of human. On the other hand necromancer might require some property of the human entity, but doesn't want to inherit the whole human class. (Muratori, 2016)

A better way of doing an entity system is composition pattern where properties are components and can be handed to any entity (see PICTURE 4) (Johansson, 2015). A game with human and orc factions might have units that can cast spells. While both the humans and orcs have these units, not all the units in the game require this functionality. With the composition pattern the spellcasting ability is attached as a component only to those units that require it, while the regular units can manage without it.



PICTURE 4: Properties handed to entities as components

Composition pattern is clearly more flexible and dynamic than the traditional way because it doesn't break down when entities require peculiar things. In games this sort of behavior is commonplace which is why it isn't a good idea to restrict functionality to the classes.



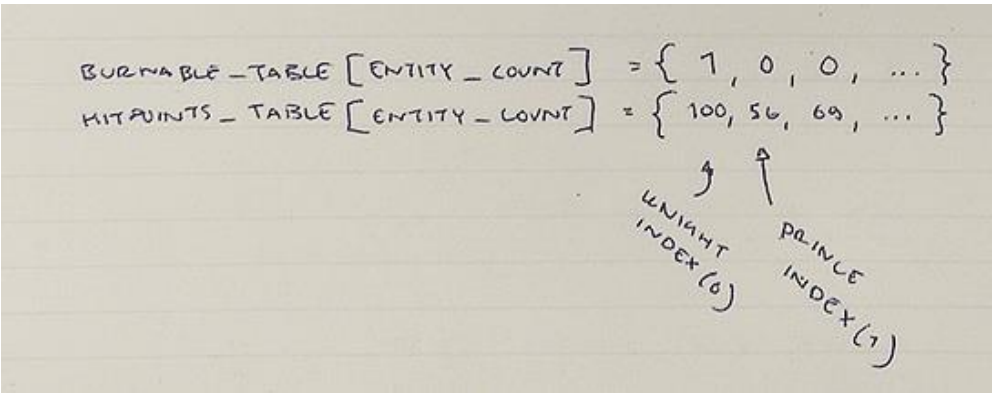
PICTURE 5: Structure of arrays

There is yet another way to handle entities. Instead of having a structure of arrays (see **Error! eference source not found.**), we can have an array of structures (see PICTURE 6). It is almost like a database with multiple tables. Entity is referred with an index. This way we can handle all the entities that contain a certain property simultaneously. This is not a typical way to manage entities.

	Necromancer	Knight	Prince
Burnable	1	1	1
Hitpoints	100	56	69
Spellcasting	1	0	1

TABLE 1: Array of structures

Consider TABLE 1. If the developer wishes to give red cheeks to each spellcasting character in their game, they can simply select everyone that has the corresponding value in the spellcasting table.

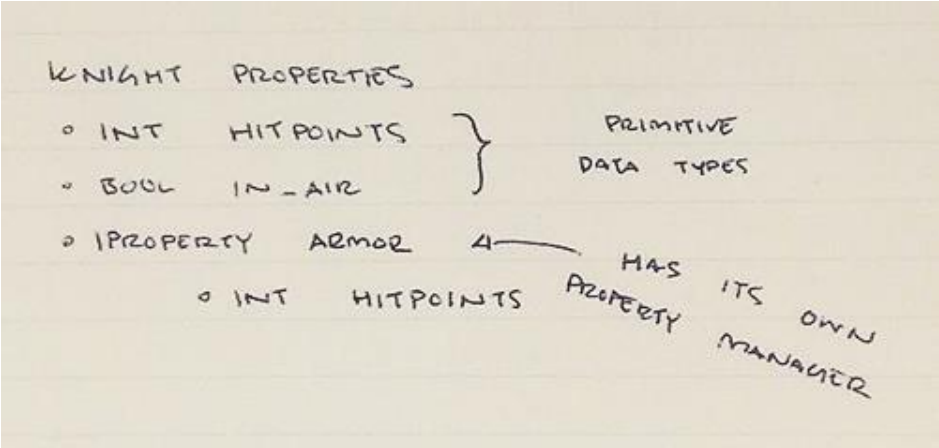


PICTURE 6: Array of structures

5.3 Property Management

Properties are what describe the entity. For an example, they tell where the entity is located, how tall it is, how large is its hitbox, if it can jump, how high it can jump, if it has hit points, and so on.

Properties don't have to be solely primitive data types. They can also be objects with their own properties. And the properties affect the parent properties if they have the same identifier. This allows the developers to create more intricate entity property management hierarchies (see PICTURE 7). (Porter, 2012, p. 6)



PICTURE 7: Nested properties

Property manager also has the flexibility to create ambiguous power-ups and similar models. For instance, when an entity collides with other entities, their properties might get merged together.

5.4 Actions

The behavior of an entity is modified with actions, such as “running” and “spellcasting”. They are defined separately and then bound to those entities that happen to require that specific behavior. This provides a flexible way to introduce behavior for entities.

Actions make changes to entity properties. For instance, “jumping” merely adds an upwards force springing the character to that direction in the game world.

In Hiillos actions are divided into standard actions, finite time actions, and instant actions. Standard actions are anything you can do by holding down a button. An example of this would be “running” action, where the location property’s X and Y values are altered while the update time causes the animation to loop.

Finite time action on the other hand is something where the player hits the button once and it performs an action from beginning to completion. For an example, “attacking” action where the character swings their sword takes a certain amount of time. The breakpoints where the frames of the animation change are set in the asset definition.

Instant actions are something that happen instantaneously; they're executed in one tick. Actions like this might include "flicking a switch" or "drinking a potion".

6 ASSET MANAGEMENT

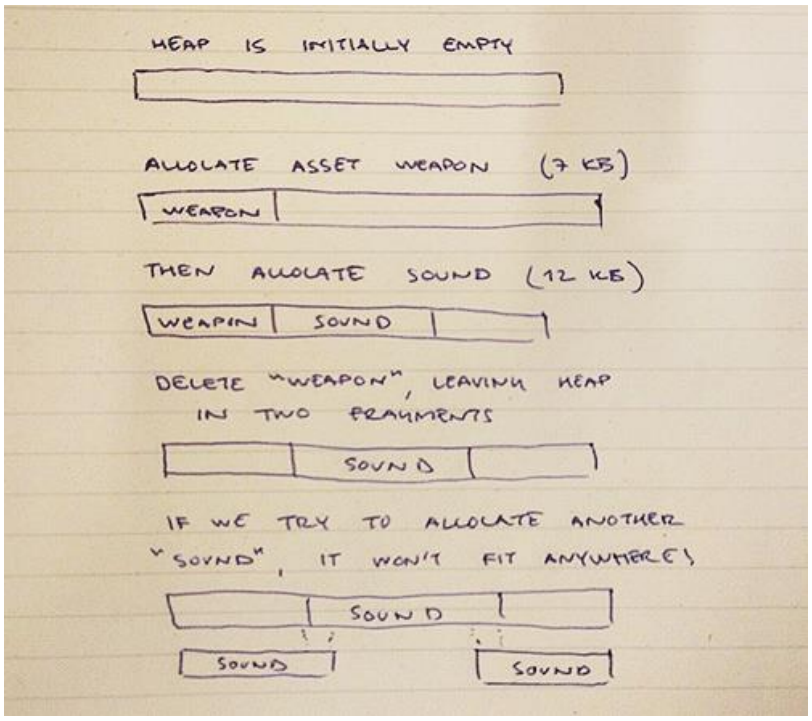
Game assets are everything not created in the program code. Typically, this includes textures, audio, and fonts. An entity never knows what assets it has. It is tagged to represent its purpose and the asset management of the engine hands it whatever assets it deems appropriate.

6.1 Asset Cache

All the assets the game happens to need are loaded into an *asset cache.* This helps to keep track of duplicates and allows the developer to know where each and every asset of the game is stored. As a result, the handling of assets is centralized and since they're all stored in list containers they can be iterated through effortlessly if need be.

Just like entity management systems, there are multiple strategies to handle asset management. The first strategy is to use fixed memory arena. At the start of the application a solid block of memory is reserved. This block is used for storing the assets. The upside of using fixed memory arena is its high performance considering the memory is not allocated from the heap because it has been reserved already. The downside is that the memory gets rather fragmented over time (see PICTURE 8). When you take into account that on modern systems games normally have hundreds of megabytes of memory reserved, fragmentation isn't much of an issue.

Another way is to use count-based strategy where there would be a reference count of each asset in the game (Gregory, 2009, p. 287). Because the references are implemented as smart pointers, whenever an asset runs out of references, the memory it was using is freed. In other words, when the last smart pointer gets destroyed, it means no entity is using the asset, and that it doesn't need to be reserving the memory any longer. This gets around the problem where the asset manager would be dependent of the entities.



PICTURE 8: Issue with fixed memory arena

Third way to do asset management is the time-based strategy, where in set intervals it is checked if the asset is in use or not. If the asset is not referenced in that particular moment, it's unloaded.

Naturally, there are other possibilities for asset management besides the aforementioned three, such as doing a variant of a garbage collection method, but out of these, Hiillos uses count-based strategy for its handy integration with modern C++ and smart pointers.

Whenever an asset is loaded, it gets loaded for an entity. The asset is returned as a C++ standard library shared pointer which has an in-built reference counting. Any amount of entities can use the same asset which keeps growing the reference count. Once no one is referencing the asset, it gets destroyed.

6.2 Asset Resolver

Entities are a collection of properties which can be always be resolved to tags. A tag is a pair of a key and a value. The key might be something like "player" or "in air" while the value determines rules to the application of the tag. The value can be an integer, integer range (from a to b), Boolean, string, or another tagged entity. (Muratori, Tag-based Asset Retrieval, 2015)

Finding an asset fitting a given circumstance is handled in the asset resolver. The functionality of the asset resolver is intimately tied to the asset tags and properties, whereby the resolver always returns a reference of an asset most closely corresponding the required case. The developer never asks for a specific asset but rather defines what is happening and the asset resolver hands them the asset that is the most fitting for the situation.

The asset resolver makes the development of a game more dynamic because the developer doesn't need to have all the game assets ready at the beginning of the project but can still program the functionality of the game as if they were. This also uncouples creating asset packs from the process since the creator doesn't need to touch the game code. The asset resolver will still return assets that represent any given situation as closely as possible. When the game assets get finished, they can be added into the project with the appropriate tags, and the asset resolver simply returns them instead of the earlier matches.

6.3 Anchors and Handles

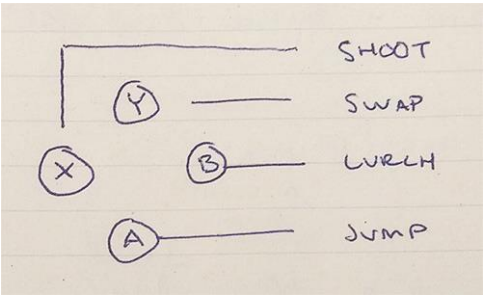
Hiillos supports definition of anchor points and handle points for assets. The idea is that the developer can define an anchor point in the base asset and then render another asset relative to that anchor point. The child asset is connected to the base asset by a handle point. For an example the developer can choose to draw weapon sprites on the player character's hand where the anchor point is located.

Anchored assets can also be detached from their parent. Because the anchored assets must have their own property managers, the developer can use that to create interesting events in the game world. For instance, if the player character is holding a gun and dies, we can detach the connection between the gun handle and the gun-holding anchor and create an effect where the character drops the weapon as it dies. Similarly, if the whole character is built from anchor points, we can make the head pop off upon a headshot.

7 CONTROL

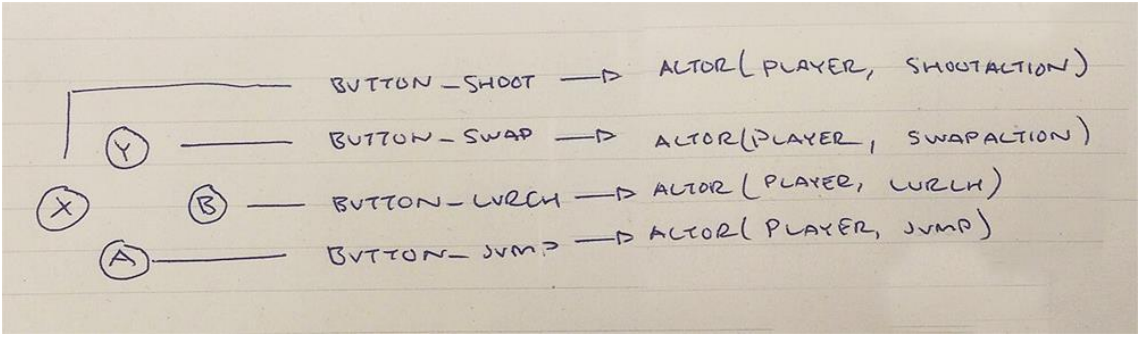
7.1 Input Handling

In every game there is a chunk of code that translates a key press into a meaningful action. In Hiillos, its simplest form means pressing a button, selecting an entity, and performing an action. This, however, is applicable to only the smallest of games where the controls have been hard-coded. (Nystrom, 2014, p. 21)



PICTURE 9: Actions bound to buttons

In more complicated games the developer can with the help of command pattern create a more dynamic way to control the input. This enables, for an example, the creation of key bindings. As Hiillos entities are already decoupled from actions this is a very logical choice for the developer.



PICTURE 10: Actions bound to commands

Hiillos doesn't provide input handling out of the box. There is no single correct way to approach the issue since large games can handle their input in a very different way compared to smaller games.

7.2 A.I. Tasks

Artificial intelligence in Hiillos is handled with a behavior tree. This a modern way of handling A.I. because, unlike state machines, it allows non-programmers to shape complex behaviors for the non-player characters.

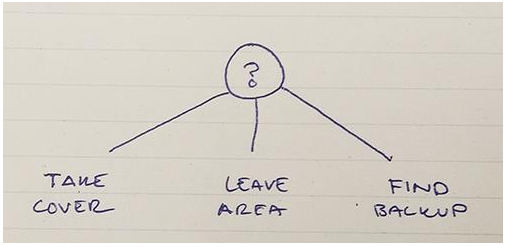
> *STATE MACHINES LACK A PROPER LEVEL OF MODULARITY. BEHAVIOR TREES INCREASE SUCH MODULARITY BY ENCAPSULATING LOGIC TRANSPARENTLY WITHIN THE STATES, MAKING STATES NESTED WITHIN EACH OTHER AND THUS FORMING A TREE-LIKE STRUCTURE, AND RESTRICTING TRANSITIONS TO ONLY THESE NESTED STATES. (SESSI, 2016)*

The behavior tree gets initialized in a scene and is updated each frame. Leaf tasks are the end nodes of the behavior tree. They describe some sort of behavior with the help of conditions and actions. The conditions ask things about the game world (e.g. what is the location of the player, how many hit points does the entity performing the task have etc.). The actions are performed if the conditions are met. If the action can be performed the leaf task returns a success.

Composite tasks offer a standardized way to describe the relationship between leaf tasks. As opposed to leaf tasks, composite tasks form the branches of the behavior tree that organize the leaf tasks. Composite tasks can be thought of as the backbone of the behavior tree that can group small parts of the behavior together to create more advanced behavior patterns. Hiillos offers three types of composite tasks: selector, sequence, and decorator.

7.2.1 Selector

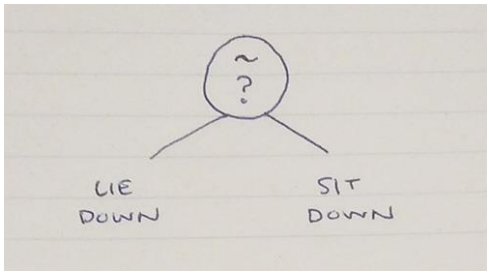
Selector is a branch task that executes leaf tasks one by one. It stops the execution if one of the leaf tasks can be successfully finished. As long as the leaf tasks return a failure, it keeps trying. If the leaf tasks run out, selector returns a failure.



PICTURE 11: Selector

7.2.2 Random Selector

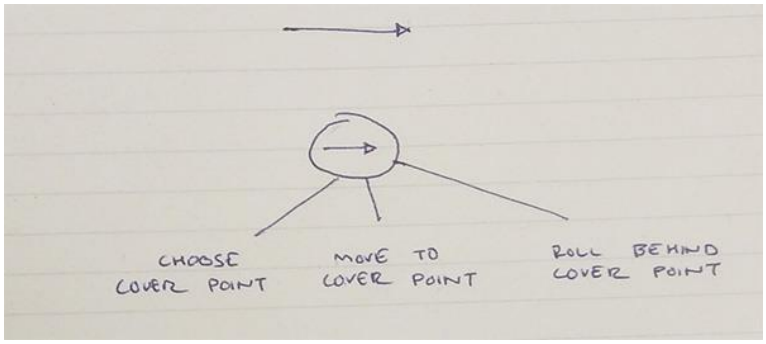
Random selector is a simple variation of the selector task. The intention is to make the A.I. more varied and interesting. While a normal selector executes its leaf tasks in a specific order which has been defined by its creator, the random selector executes tasks in a random order. E.g. if a fox wants to rest, it can either go sit down or lie down. This behavior can be created by adding a random selector with two leaf tasks: “sit down” and “lie down”.



PICTURE 12: Random selector

7.2.3 Sequence

Sequence is a branch task that executes the leaf tasks in a sequence. As opposed to the selector, sequence stops the execution if any of the leaf tasks returns a failure. As long as the leaf tasks keep returning success, the execution continues. If the leaf tasks run out, the sequence returns a success.

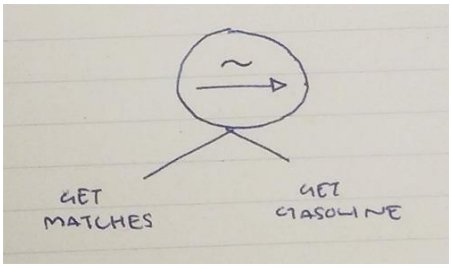


PICTURE 13: Sequence

Sequence describes a series of tasks that need to be executed. For instance, in PICTURE 11, the task tagged as “take cover” could be a sequence. In order to take cover the character first needs to choose a cover point, move in its direction, and finally roll behind it. If one of the steps in the sequence fails, the whole sequence fails. If the selected cover point cannot be reached, then “take cover” fails. Only when all the leaf tasks in a sequence are executed can the whole sequence be considered completed.

7.2.4 Random Sequence

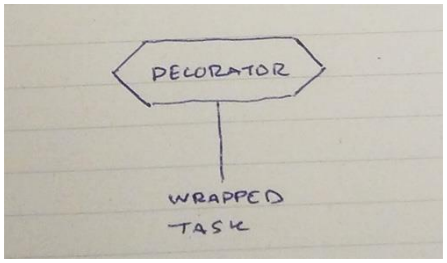
Similarly to selector, there is a random variation of the sequence that executes its leaf tasks in a random order. This is useful in certain scenarios. For instance, consider wanting to give an entity a task to burn something. In order to be able to complete this task the entity needs gasoline and matches. The order in which the entity gets these items, however, is irrelevant. This task can be implemented by creating a random sequence with two leaf tasks: “get matches” and “get gasoline”.



PICTURE 14: Random sequence

7.2.5 Decorator

The name “decorator” derives from object-oriented programming. Decorator pattern refers to a class that wraps another class by altering its behavior. If the decorator has the same interface as the class that it wraps, the rest of the application doesn’t need to know how to handle the decorator and the original class separately.



PICTURE 15: Decorator

In the context of a behavior tree the decorator has one leaf task that alters the behavior of the task in some way. The decorator task can be thought of as a composite task with one leaf task.

Hiillos offers a collection of decorators:

AlwaysFail	Whether the wrapped task succeeds or not is irrelevant – its status is always: failure.
AlwaysSucceed	Whether the wrapped task succeeds or not is irrelevant – its status is always: success.
Invert	Succeeds if the wrapped task fails, fails if the wrapped task succeeds.
Limit	Regulates how many times the wrapped task can be executed. Assures that the A.I. doesn't get caught in a loop.
Repeat	Repeats the wrapped task x times, potentially infinitely. Returns a success when it reaches the set execution amount.
UntilFail	Executes the wrapped task until it fails. At that point the decorator returns a success.
UntilSuccess	Executes the wrapped task until it succeeds. At that point the decorator returns a success.

TABLE 2: Decorators in Hiillos

8 EVENT MANAGEMENT

In this world we have entities built from properties. These properties change when actions occur. Sometimes an action can have a side effect that influences another entity's behavior or properties. Sometimes it is important that the entity or the scene knows that something has happened.

For an example, the game might have an achievement system that keeps a track of certain in-game property changes. If the player does something in a certain amount of time they get rewarded by an achievement.

At this point it becomes evident that some sort of event system helps in the development of this sort of functionality. As is typical to the observer pattern, the subjects in the game broadcast changes in desired in-game properties and the observer receives this information. (Salmi, 2004)

9 SUMMARY

The tendency for complex, heavy, monolithic game engines, such as Unity, seems to be dominant in the present-day game industry. We believe game developers should have control over which tools and libraries they want to use to build their game.

Hiillos itself is not a game engine but rather a game foundation library that offers game developers a collection of tools of which they can choose the ones they find relevant to their game project. It helps with structuring the code, managing resources, and creating sophisticated artificial intelligence behaviors.

The core principles when building Hiillos-powered games are: blind entities, actions alter properties, and scenes define scope. Entities should not know about their behavior nor how they are rendered. They only know their properties which are altered by the actions. Scenes are where entities act.

The way Hiillos handles assets uncouples them from the game code. The assets aren't called directly because this would mean that the developer would have to know which assets they have available.

Hiillos provides a behavior tree functionality and a concept of tasks for the developer which allows them to assign tasks to the A.I. which then in turn perform actions based on their tasks.

REFERENCES

Cornut, O. (2016, May 6th). *dear imgui*. Retrieved May 11th, 2016, from GitHub: https://github.com/ocornut/imgui

Ewald, M. (2012, November 7th). *Game State Management*. Retrieved May 25th, 2016, from Cygon's Blog: http://blog.nuclex-games.com/tutorials/cxx/game-state-management/

Gregory, J. (2009). *Game Engine Architecture.* A K Peters/CRC Press.

Johansson, M. P. (2015, September 20th). *Composition over Inheritance*. Retrieved May 13th, 2016, from Medium: https://medium.com/humans-create-software/composition-over-inheritance-cb6f88070205#.4g2ghsrvo

Muratori, C. (2015, June 15th). *Tag-based Asset Retrieval*. Retrieved May 25th, 2016, from YouTube.com: https://youtu.be/7g79J2aMTUM

Muratori, C. (2016, May 5th). *Handmade Hero Day 277 - The Sparse Entity System*. Retrieved May 13th, 2016, from YouTube: https://www.youtube.com/watch?v=wqpxe-s9xyw

Nystrom, R. (2014). *Game Programming Patterns.* Seattle: Genever Benning.

Porter, N. (2012). *Component-based game object system.* Ottawa: Carleton University.

Salmi, Y. (2004, September 16th). *Simple Event Handling*. Retrieved May 25th, 2016, from Gamedev.net: http://www.gamedev.net/page/resources/_/technical/game-programming/simple-event-handling-r2141

Sessi, D. (2016, February 26th). *Behavior Trees*. Retrieved May 23rd, 2016, from Github: https://github.com/libgdx/gdx-ai/wiki/Behavior-Trees#core-concepts