Punity – a hierarchical and dynamic system for games

Ștefan-Alexandru, Zamfir
Faculty of Automatic Control and Computer Engineering
Technical University of Iași
Iași, România
stefan-alexandru.zamfir@student.tuiasi.ro

***Abstract*—Nowadays, developing a complex game requires a versatile way of managing resources. I propose a simple hierarchical system that supports delayed callbacks, dynamic inserting and removing of entities and provides a completely abstracted way of interacting with the entities. I also give a real example of a game built with this system in C++ which is to be run on a Raspberry Pi Pico microcontroller to showcase its versatility and performance.**

***Index Terms*—software applications, embedded systems, algorithms and programming**

I. Introduction

When thinking about developing a game, it is normal to start by creating an endless loop which governs over an unordered system (Fig. 1).

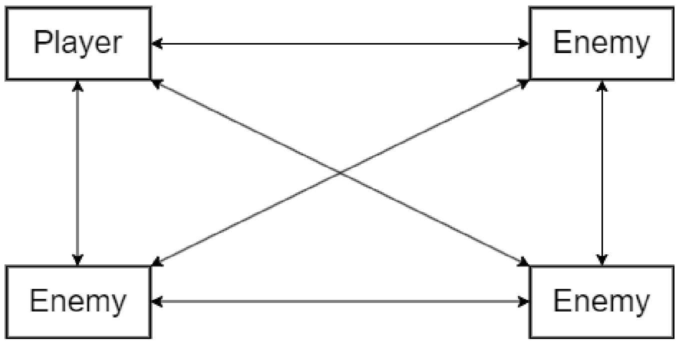


Fig. 1. An unordered system.

For extremely simple games composed of no more than a handful of entities, this system works fine. However, it is not ideal for scalability. In order for a system to be scalable, it must manage entities' dependencies in a way that is modular – which implies that any change to the code of one entity will impact a minimal amount of code in other entities. This cannot be achieved since everything is interconnected in this system.

For the above example, if I wanted to add a way of increasing enemies, I must reconsider how I collect references inside the player and how enemies reference each other, updating the logic of all entities.

A simple improvement would be to create an almost-hierarchical system (Fig. 2).

Using managers gives better modularity since managers act as a grouping mechanism for entities and command them

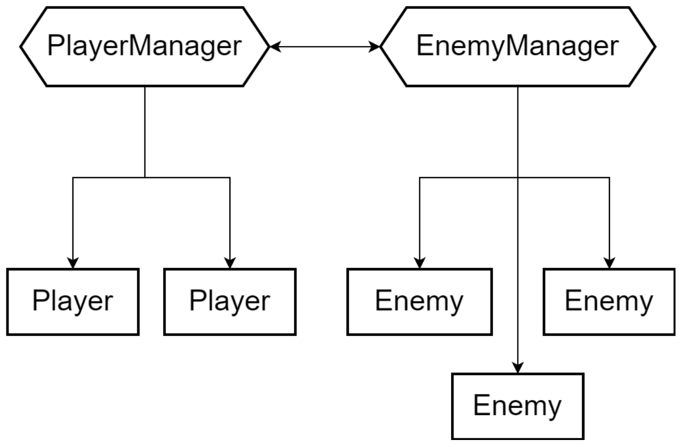


Fig. 2. An almost hierarchical system.

directly by providing a bidirectional feedback system, while also communicating with other managers. Most of the time, simple games will implement this architecture. However, this is not completely scalable since references are difficult to keep track of. For example, consider the options for switching a level:

- Method 1: disable all entities one by one;
- Method 2: implement a function for each manager that disables the reference they keep track of and then disable the managers themselves (repeatable code).

This quickly becomes suboptimal if you have a lot of systems and also maintain dynamic entities.

The system that I propose is completely tree-based/hierarchical and solves the issues of scalability by enforcing strict and intuitive rules of building behavior in entities (Fig. 3).

In order to keep the memory footprint as small as possible, C++ was chosen as the implementation language. The example code will also be written in C++. It supports the Raspberry Pi Pico SDK natively and gives complete control over memory as the board has only 264kB of RAM.

In section II, structure and functionalities will be discussed and in section III, design. Finally, in section IV a real-world implementation of the engine will be presented.

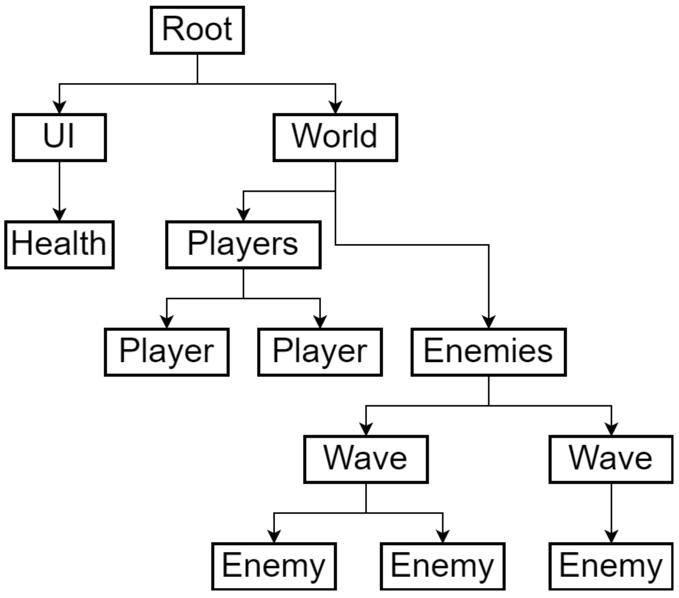


Fig. 3. A fully hierarchical system.

II. Structure and Functionalities

A. Before starting

Following up are only the rules of the building blocks that make the system. After defining them, I will start explaining how to use them effectively.

B. Entities

It is useful to imagine that an entity is no more than an empty shell. It has no purpose other than to exist in the tree structure of the system. This means that there exists no "Player" entity or "Enemy" entity – these are just their names. What acts are their components, which are synonymous with behavior (Fig. 4).

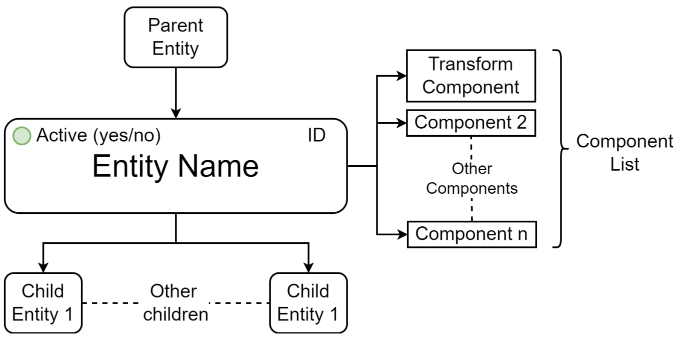
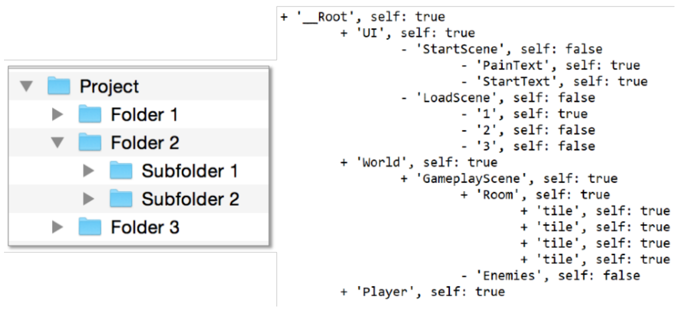


Fig. 4. Structure of entity.

The data an entity stores is as follows:

- Name;
- Unique id;
- List of components;
- A transform component (which is included in the list of components by default);
- Parent entity;
- Children entity;
- Active flag.

Entities form a tree structure which is similar to how folders work in usual File Systems (Fig. 5).



```
+ '__Root', self: true
        + 'UI', self: true
                - 'StartScene', self: false
                        - 'PainText', self: true
                        - 'StartText', self: true
                - 'LoadScene', self: false
                        - '1', self: true
                        - '2', self: false
                        - '3', self: false
        + 'World', self: true
                + 'GameplayScene', self: true
                        + 'Room', self: true
                                + 'tile', self: true
                                + 'tile', self: true
                                + 'tile', self: true
                                + 'tile', self: true
                        - 'Enemies', self: false
        + 'Player', self: true
```

Fig. 5. A comparison between folders and the hierarchial system presented.

Structuring rules are as follows:

- Deleting an entity deletes all sub-entities analogous to how deleting a folder deletes all sub-folders.
- Disabling an entity makes all sub-entities inactive but does not disable them analogous to how hiding a folder makes sub-folders unreachable but does not hide them. To be clear, it means that if an entity is enabled but their parent is not, the entity is "inherently" unable to participate until the parent is enabled again.
- Components are like the files of a folder. Components cannot have sub-components similar to how files cannot have sub-files and are intrinsic objects.

Entities can be created, deleted, moved, disabled or enabled as demanded by components' logic.

C. Components

A component is an abstract object that acts upon the entity it is attached to. Components are what breathe "life" into entities (Fig 6).

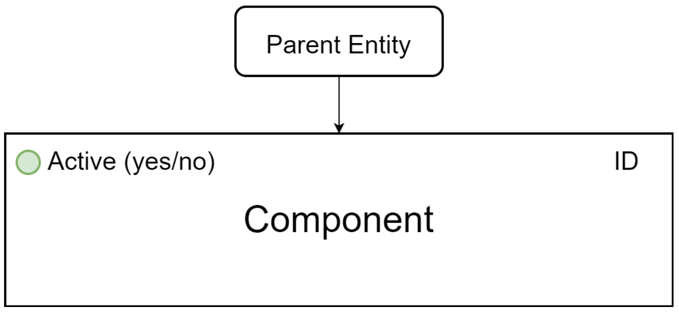


Fig. 6. Structure of component.

The data a component stores are:

- Unique id;
- Parent entity;
- Active flag.

There are some rules that apply for components:

- There is no limit to the number of components an entity can have;
- A component has only one parent;
- A component can only have an entity as a parent.

Components function using event triggers. Every component will inherit a base component which defines all the event triggers in order for the engine to make use of polymorphism (Fig. 7) .

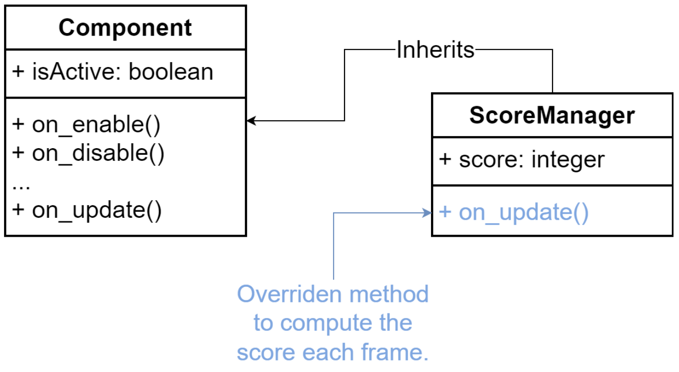


Fig. 7. Example of method overriding.

The base component has the event triggers defined but without any logic. This way, if you do not override a trigger you can safely assume it won't act without notice (Table I).

TABLE I
COMPONENT EVENT TRIGGERS

Event Trigger	When it's called
void on_enable()	Each time the component is enabled.
void on_disable()	Each time the component is disabled.
void on_destroy()	When the component is destroyed.
void on_start()	When the engine starts.
void on_update()	Each frame.
Collision-related	
void on_start_collision(Collider*)	When you first touch another collidable.
void on_collision(Collider*)	Each frame you touch another collidable.
void on_end_collision(Collider*)	When you stop touching another collidable.

A number of components must be implemented which interact with the engine in a special way. This happens either because you need the engine to sort them in a special way (e.g. layering sprites) or you need them to trigger default events (e.g. colliders are part of the collision events).

The components that are already implemented are as follows (regarding a 2D world):

- BoxCollider2D, CircleCollider2D: used to enable an object's collision;
- SpriteRenderer: used to render world-space sprites on the screen;
- UIRenderer: used to render screen-space sprites on the screen;
- Transform: used to keep track of an entity's coordinates in world-space and to recursively reflect its changes to the entity's subtree.

As an example, let us make a component which changes an entity's location when collided by some other entity. This will use an implemented C++ engine which I will present later on this paper but with minor changes to names for the sake of readability.

```
class TranslateOnHit : public Component {
    void on_start_collision(Collider* other) override {
        // Get parent entity of component
        Entity* parent_entity = this->get_entity();

        // Get Transform component of entity
        Transform* entity_transform = parent_entity->
            ↪ get_component<Transform>();

        // Translate entity by 10 pixels on the Y axis
        entity_transform->translate({0, 10});
    }
}
```

D. The Root Entity

At least one entity exists at all times. I call that entity the Root entity. The Root entity:

- Cannot be removed;
- Can only be a parent entity, meaning that no entity may have the Root as a child;
- Cannot be renamed;
- Cannot be disabled.

The Root entity is similar to the root of a tree structure. Each time an entity is created but no parent is specified, it should be attached to the Root entity.

E. Invokers

It is common that an user may want to call a function after a certain time has passed. For example, you may want to change an entity's name after 1 second a button has been pressed. This can be achieved using some variables and checking whether the time to call the function has been reached.

However, this not only isn't elegant, it is also not scalable. Nesting such calls would take an enormous amount of lines and readability would suffer.

Invokers are the solution. I have managed to make them work in C++ which is firm when it comes to functions, especially methods (Fig 8).

If an invoker is still inactive but the host entity was destroyed, the invoker must be destroyed as well.

Here is a code snippet of how an invoker could be used:

```
class SomeComponent : public Component {
    void on_start_collision(Collider* other) override {
        // When hit by another entity, call function_to_call after 5
            ↪ seconds.
        new Invokable<SomeComponent>(
            // Function to call
            &SomeComponent::function_to_call,
            // Pointer to instance (need to know on which object
                ↪ to call the function)
```

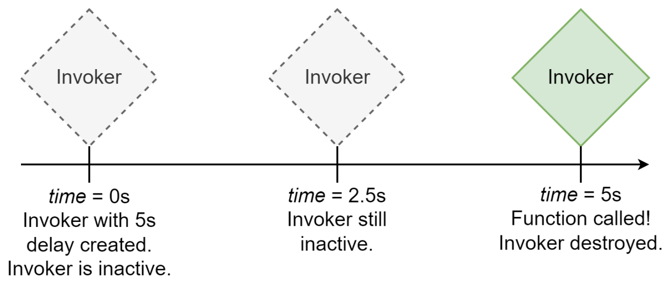


Fig. 8. Showcase of an invoker that calls a function with a 5 second delay.

```
            this,
            // Delay in seconds
            5.0f,
            // Id of entity for removal in case entity is destroyed
            get_entity()->get_id()
        );
    }

    void function_to_call() {
        // Some Logic
    }
}
```

Since I chose to write this engine in C++, invokers are going to be limited without more extensive work. The version made by me in C++ only works for **void** (**void**) type functions (However, any other type can be implemented – it's just I couldn't generalise it). Invokers are a section which could greatly benefit from some extending. Of course, if the engine were implemented for example in Python, it would natively support any function type. However, the idea of an invoker is the main point here.

F. Engine

In principle, the engine is a simple, endless loop that updates the states of components, renders the results on the screen and optionally pauses for a certain amount of time (Fig. 9). In my case, the engine maintains a list of all currently active entities and processes that list every frame.

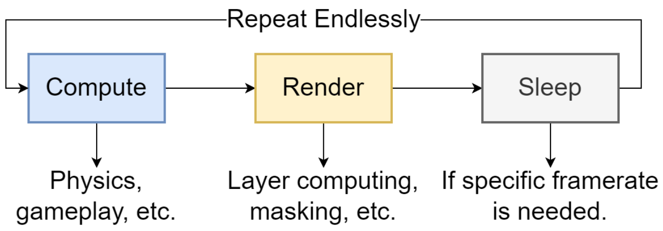

Repeat Endlessly
Compute
Render
Sleep
Physics, gameplay, etc.
Layer computing, masking, etc.
If specific framerate is needed.

Fig. 9. The engine pipeline.

In practice, things can get more complicated. The main reason is order of execution. My approach is similar to Unity's [1].

What happens if I destroy an entity and then a component wants to act on that destroyed entity? How do I know which entity is processed first? What happens if I create a sub-entity inside on_destroyed()?

To solve this, I've defined an order of execution which guarantees that you will not have to deal with invalid entities at any time (Table II).

TABLE II
ORDER OF EXECUTION

1. Check which entities are disabled/destroyed and remove them and their invokers from the active entity list. Keep a list of the entities which should be destroyed for the next step. It is safe to notify the components with on_destroyed().
2. Safely delete the entities marked for destruction entities, releasing their memory. The new frame starts from this point on.
3. Load the background or clear the screen.
4. Loop through all the entities and call on_update() on their components. It does not matter if an entity is marked for destruction or is disabled at this stage, it will be updated anyways. This means that destroying is delayed until the next frame. This is also the stage in which the engine collects important component info such as Colliders and Sprites.
5. Loop through all the invokers and solve them.
6. Compute collisions. They are partitioned into static and dynamic colliders as an efficiency measure.
7. Draw World Sprites and then UI Sprites. Both groups are layered, so an ordered data structure (std::multiset) must be used to store them.
8. Send the frame to the screen controller.
9. Sleep the remaining time if a specific frame rate is wanted.
10. Go back to step 1.

This ensures that deletion is done completely out of sight of component code. By separating event dispatching from memory deletion, components cannot interact with possibly invalid entities.

An important note is that entities are updated in an undefined order. If necessary, entities could be updated starting from the Root entity and recurse to the bottom if such need was critical.

The engine also updates the Time util necessary for other components to keep track of time. For example, Time::
↪ delta_time returns the seconds that have passed since the last frame was finished and is used heavily in time-dependant physics computation such as moving.

III. DESIGN

The whole engine is aimed towards easily achieving the SOLID [2] principles. I will present a practical implementation of the *Single-responsibility* principle as an example use case of the engine's features.

Single-responsibility in Punity

This is a style of writing logic for components that aims to modularize an entity's complex behavior into smaller, simpler behaviors.

For example, consider that I were to create a player entity which is controlled by a joystick, swings a sword and collides with physical structures. I could make a component which encapsulates all of that. However, it is also probable that in the future, I may want the player to handle different kinds of weapons. I may want to control the player with some other device. I may also want to create enemies that swing their swords at the player (Fig. 10).

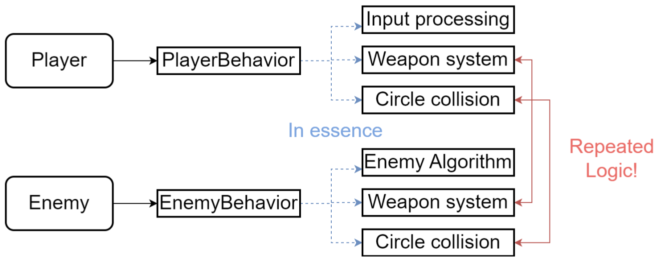


Fig. 10. Repeatable code.

It would be a waste of time to repeat writing the same behavior for different components (e.g. weapon of player and weapon of enemy). I should not make a particularized, all-in-one component for each and every entity. Knowing that an entity can have multiple components, I should aim to split complex behavior in simple, general and repeatable behaviors (Fig. 11).

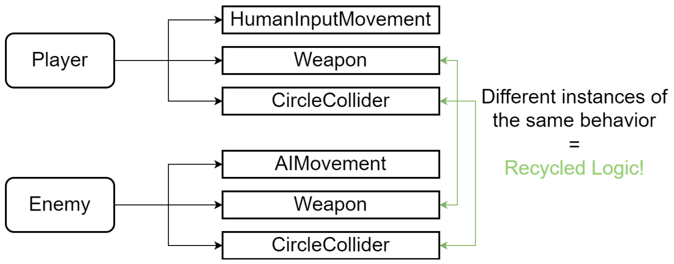


Fig. 11. Component reusability.

My player would now have:

- HumanInputMovement component;
- Weapon component;
- CircleCollider component;

While the enemy would have:

- AIMovement component;
- Weapon component;
- CircleCollider component;

The components are recycled, linked and easily updatable now and they each have one single responsibility. If I were to have a bug in my algorithm for CircleCollider, fixing it once would fix it for each and every entity that uses it. On the other hand, having every entity implement the same algorithm would be time consuming to fix.

IV. Portability on Raspberry Pi

I've developed this engine in C++ [3] but also made a game to showcase its features and workflow (Fig. 12). The engine was intended to be used with the Raspberry Pi Pico platform which has only 264kB of RAM and an Arm Cortex-M0+.

Everything was built from the ground up, including the screen, joystick and button drivers. I have abstracted them in such a way that such drivers could be easily ported to other platforms.

The game has a main menu and 3 levels each with 3 stages. In each stage different kinds of enemies move and shoot at you and you must eliminate them before you are gone. The game features a weapon system, a smart aiming system which targets enemies that aren't behind a wall, a pickup system and many more things.

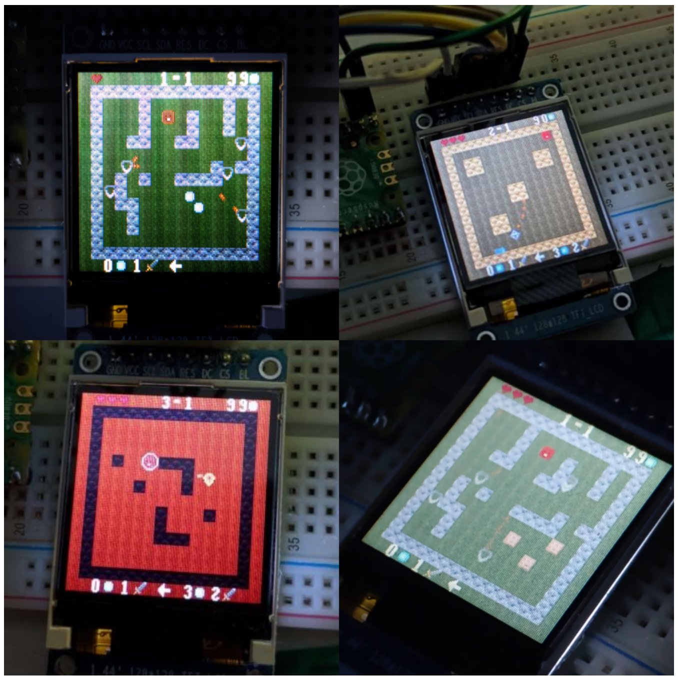


Fig. 12. The game I developed, “Pain”.

The project is split into two parts:

- Engine code: Components and Utils that make up the engine which can be extended.
- Game code: Components and assets which use the engine and build the actual game.

A. Engine code

As I said, I split up Engine code into Components and Utils.

These are the provided components (Fig. 13) which can be attached to an entity and are special because of their relationship with the engine as described in Table II:

- PBoxCollider.h and PCircleCollider.h: They give a Box or a Circle-shaped “hitbox” to the entity.
- PSpriteRenderer.h and PUISpriteRenderer.h: Given a sprite, the engine will render it on the entity's center. The difference is that the UI is invariant to the entity position.
- PTransform.h: Describes position in world space and implements the relativity principle mentioned in section II-B.

Utils are utilities which are essential to usual game development use cases:

- PCollisionComputation.h: used for writing/extending collision logic.
- PError.h: used for error logging.
- PInvokable.h: presented earlier, used for invoking methods.
- PMath.h: useful mathematical functions, usually involving Vectors.

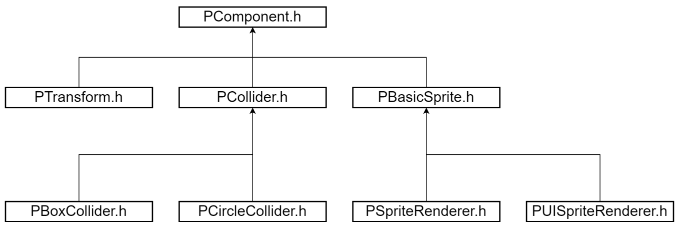


Fig. 13. Components of the Punity Engine and how they are made.

- PRandom.h: pseudo-random number generator. My platform did not have reliable randomness.
- PTime.h: for keeping track of elapsed time and frame time.
- PVector.h: for representing 2D vector spaces.
- PButton.h and PJoystick.h: input for the game.

Let's compare my engine to Unity's. Scripting using this engine looks something like this:

```
class MyComponent : public Punity::Components::PComponent {
    void on_update() override {
        std::cout << "Hello!\n"; // Prints 'Hello!' every frame.
    }
};

// Make an entity
auto my_entity = Punity::PEntity.make_entity("My_entity", true);

// Add a component to that entity
my_entity.add_component<MyComponent>();

// Get the component
my_entity.get_component<MyComponent>();
```

While not exactly equivalent (Unity has an incredibile visual editor and it's C#, not C++) this is how scripting is done in Unity:

```
using UnityEngine;
using System.Collections;

public class ExampleBehaviourScript : MonoBehaviour
{
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.R))
        {
            GetComponent<Renderer>().material.color = Color.
                ↪ red;
        }
    }
}
```

They are very similar. Both inherit a Base class which represents a component and both override events. Both also make use of generics such that users intuitively interact with the components.

This is my second attempt at making an engine and redesigning all over again actually led to converging with Unity's philosophies, which are an industry standard.

B. Screen and memory optimizations

There are some optimizations regarding how the screen is rendered since my memory was limited to 264kB. My screen has a 128 by 128 resolution and each color has 2 bytes of information (RGB565). This makes up to 32kB just to store a full picture. That's not good news.

It's important to note that sprites have a space complexity of $O(width \cdot height)$ bytes. I use an upper bound since single-coloured sprites should really be a single byte if you care about space.

I usually have square sprites, so that makes the space complexity a quadratic. With this in mind, I've chose to make sprites as small as possible while not sacrificing visual fidelity. The background is actually a repeated tile, using only 64 bytes instead of 32kB.

However, there's another problem with the screen – it is really slow. If I were to simply send the sprites, the screen would tear constantly as it can't keep up with the rate I am sending the frames.

The solution comes by using buffers and a change matrix (Fig. 14).

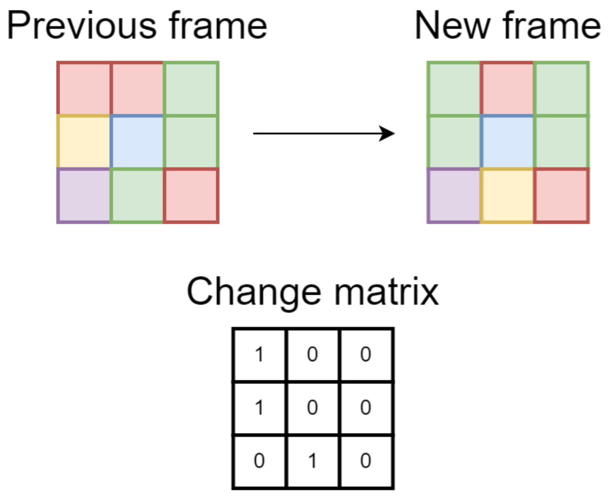


Fig. 14. Chage matrix sent to the screen.

The previous frame is always stored and the next frame is computed. Then, the engine compares which pixels have changed after all computation, sending the screen only the minimum amount of pixels.

Using this technique, frame rate has improved significantly to the point it's a constant 60 frames per second. This costs around 80kB, but without it the experience would be a lot worse.

C. Collider optimizations

Each of my levels has around a hundred entities with colliders attached at any time. They are:

- Walls;
- Destructible boxes;
- Bullets from enemies and player;
- Player;
- A few enemies;
- Energy/shield/heart pickups;

It is necessary to compute collisions between entities in order to have a functioning physics system. A naïve approach would loop through all colliders for each and every collider and solve the collision.

However, this approach has a quadratic time complexity and may be expensive depending on what kind of collisions you compute.

Does it make sense to compute collisions between walls, for example? They always stay fixed in place, so why should I check if a wall pushes another?

This is why I've introduced the notion of static and dynamic colliders, together with common and trigger colliders. Figure 15 describes a process which corrects the dynamic's entity position based on intersection with the static collider. The dotted red circle is the next frame's position and is corrected by the engine.

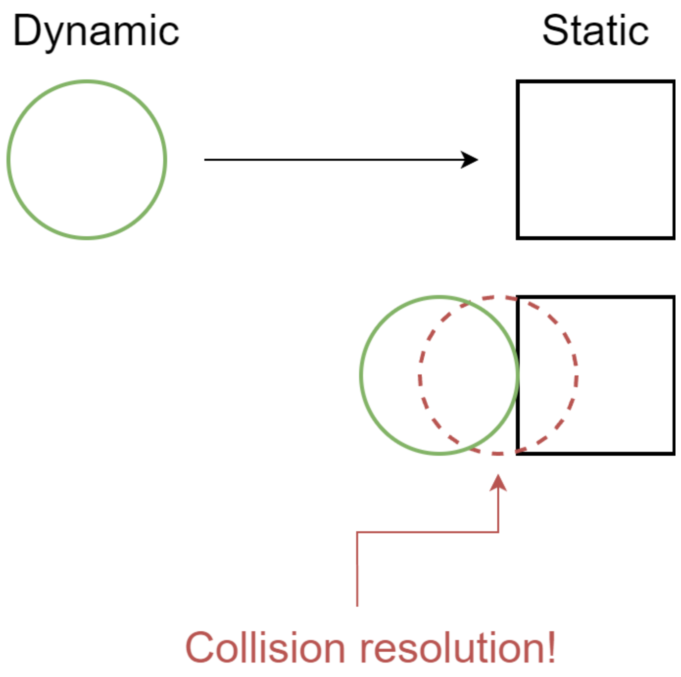


Fig. 15. Example of collision resolution.

A static collider participates in collision events. That means the events on_collision(), on_start_collision() and on_end_collision
↪ () are called accordingly. A dynamic collider, however, can also have its position updated. Basically, it has a response to collisions while statics do not.

A trigger only checks for intersections, so when a dynamic entity encounters a static trigger the dynamic entity is not translated.

It would be intuitive to think that any moving entity should have a dynamic collider, but that is wrong:

1) Bullets don't need any physicality since they vanish on collisions, so they are sliding, static triggers.
2) Enemies don't need to be dynamic because their paths are algorithmic and will always avoid terrain.
3) Pickups only need to be static triggers just like the bullets.

This leaves the player as the sole dynamic entity, reducing the time complexity to linear time.

V. Conclusion

By utilizing a tree-based structure and enforcing strict, intuitive rules for building behavior in entities, the proposed system demonstrates excellent scalability and versatility. The choice of C++ as the implementation language ensures minimal memory footprint and native support for the Raspberry Pi Pico SDK, making it an ideal solution for resource-constrained environments such as the Raspberry Pi Pico microcontroller with only 264kB of RAM.

Further work may expand the colliders, add more components, support more displays and inputs.

References

[1] Unity 2023, *Unity - Manual: Order of execution for event functions*, accessed 5.04.2023, link.
[2] Martin, Robert C. 2003, *Single Responsibility Principle*, accessed 5.04.2023, link.
[3] Zamfir, Ștefan A. 2022, *Punity*, accessed 5.04.2023, link.