Small Game Engines, Big Lessons: PolyEngine & Pill Engine

Jakub Duchniewicz

Everything Open 2026

Ngunnawal (Canberra, ACT) 23.1.26

Small Game Engines, Big Lessons: PolyEngine & Pill Engine

Jakub Duchniewicz

Everything Open 2026

Ngunnawal (Canberra, ACT) 23.1.26

Hi, I'm Jakub and welcome to my talk titled Small Game Engines, Big Lessons. Just making sure you know you are in the proper room!

Acknowledgement

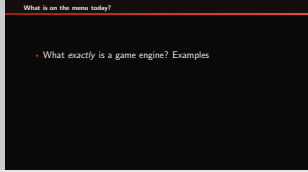
I would like to acknowledge the Ngunnawal people as the traditional owners of these lands. I feel very privilleged to be here and present to you pieces of my knowledge on these unique grounds.

Acknowledgement
I would like to acknowledge the Ngunnawal people as the traditional owners of these lands. I feel very privilleged to be here and present to you pieces of my knowledge on these unique grounds.

To follow online

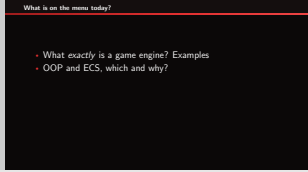
To follow online

- What *exactly* is a game engine? Examples



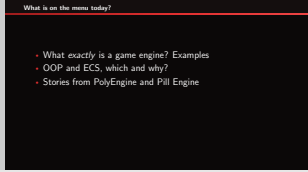
Let's maybe start with asking how many of you have some experience making games? And how many have worked with custom engines or contributed to one? Here is rough outline for today and we have a lot to cover so let's start!

- What *exactly* is a game engine? Examples
- OOP and ECS, which and why?



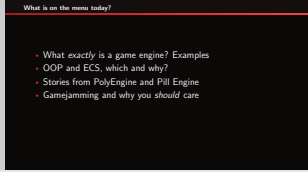
Let's maybe start with asking how many of you have some experience making games? And how many have worked with custom engines or contributed to one? Here is rough outline for today and we have a lot to cover so let's start!

- What *exactly* is a game engine? Examples
- OOP and ECS, which and why?
- Stories from PolyEngine and Pill Engine



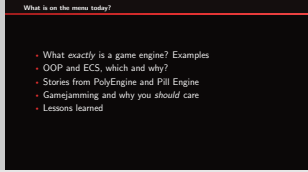
Let's maybe start with asking how many of you have some experience making games? And how many have worked with custom engines or contributed to one? Here is rough outline for today and we have a lot to cover so let's start!

- What *exactly* is a game engine? Examples
- OOP and ECS, which and why?
- Stories from PolyEngine and Pill Engine
- Gamejamming and why you *should* care



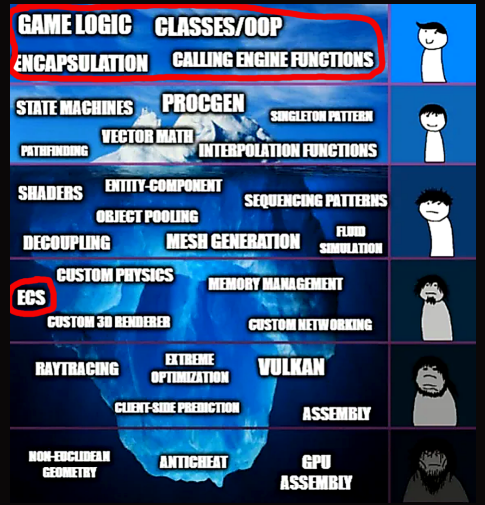
Let's maybe start with asking how many of you have some experience making games? And how many have worked with custom engines or contributed to one? Here is rough outline for today and we have a lot to cover so let's start!

- What *exactly* is a game engine? Examples
- OOP and ECS, which and why?
- Stories from PolyEngine and Pill Engine
- Gamejamming and why you *should* care
- Lessons learned

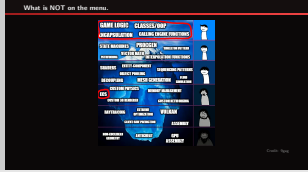


Let's maybe start with asking how many of you have some experience making games? And how many have worked with custom engines or contributed to one? Here is rough outline for today and we have a lot to cover so let's start!

What is NOT on the menu.



Credit: 9gag



Before we descent into the depths let's examine the iceberg that game development is. Unless you are writing your own engine or your game aims to be the next GOTY, you mostly touch the 2 top layers. If you think this is the whole iceberg, I am sorry to disappoint you, we have even more topics that might be game-specific!

Embedded and game engine programmer. Worked with 5G/LTE, audio/video processing, networking protocols and custom Hardware & FPGAs.

Co-founder @ Sticky Piston Studios.

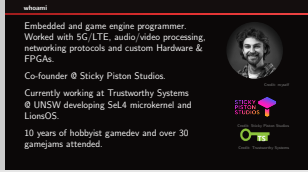
Currently working at Trustworthy Systems @ UNSW developing SeL4 microkernel and LionsOS.

10 years of hobbyist gamedev and over 30 gamejams attended.

Credit: myself

Credit: Sticky Piston Studios

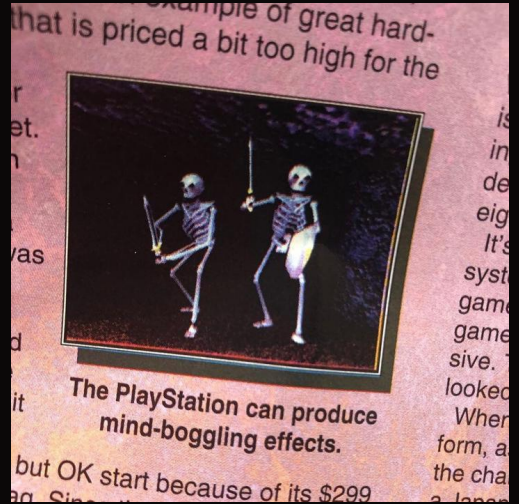
Credit: Trustworthy Systems



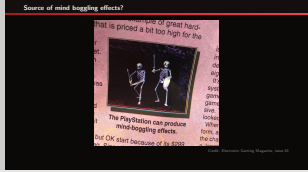
Emphasize embedded + game engine background; mention seL4/LionsOS as “high-assurance OS work”; keep it to 15s.

What is a game engine?

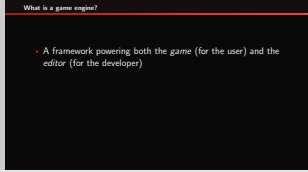
Source of mind boggling effects?



Credit: Electronic Gaming Magazine, issue 61

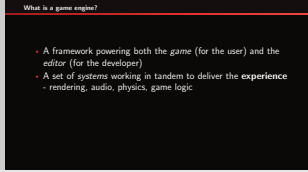


- A framework powering both the *game* (for the user) and the *editor* (for the developer)



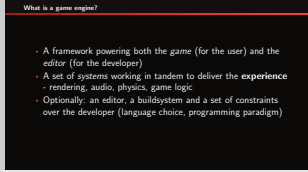
So what exactly is a game engine? It is a framework that has multiple systems working in tandem to present the world to the player and the game developer. It often comprises custom build systems, and opinionated SDK along with other goodies

- A framework powering both the *game* (for the user) and the *editor* (for the developer)
- A set of *systems* working in tandem to deliver the **experience**
  - rendering, audio, physics, game logic



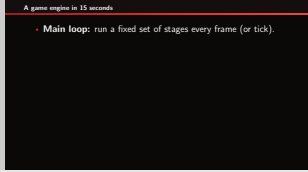
So what exactly is a game engine? It is a framework that has multiple systems working in tandem to present the world to the player and the game developer. It often comprises custom build systems, and opinionated SDK along with other goodies

- A framework powering both the *game* (for the user) and the *editor* (for the developer)
- A set of *systems* working in tandem to deliver the **experience** - rendering, audio, physics, game logic
- Optionally: an editor, a buildsystem and a set of constraints over the developer (language choice, programming paradigm)



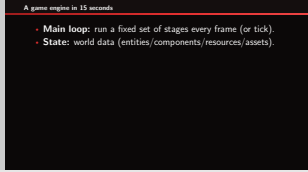
So what exactly is a game engine? It is a framework that has multiple systems working in tandem to present the world to the player and the game developer. It often comprises custom build systems, and opinionated SDK along with other goodies

- **Main loop:** run a fixed set of stages every frame (or tick).



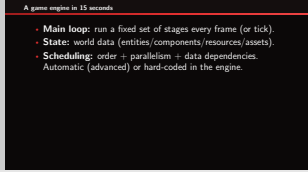
Here you can see a pseudocode of a main loop - our heart of the game engine that beats every tick. The systems update the state of the world, and it's done differently depending on the paradigm of which I will tell you later. The order of how it's done can be fixed or dynamic depending on the complexity of the engine. Here you can see it hardcoded.

- **Main loop:** run a fixed set of stages every frame (or tick).
- **State:** world data (entities/components/resources/assets).



Here you can see a pseudocode of a main loop - our heart of the game engine that beats every tick. The systems update the state of the world, and it's done differently depending on the paradigm of which I will tell you later. The order of how it's done can be fixed or dynamic depending on the complexity of the engine. Here you can see it hardcoded.

- **Main loop:** run a fixed set of stages every frame (or tick).
- **State:** world data (entities/components/resources/assets).
- **Scheduling:** order + parallelism + data dependencies. Automatic (advanced) or hard-coded in the engine.



Here you can see a pseudocode of a main loop - our heart of the game engine that beats every tick. The systems update the state of the world, and it's done differently depending on the paradigm of which I will tell you later. The order of how it's done can be fixed or dynamic depending on the complexity of the engine. Here you can see it hardcoded.

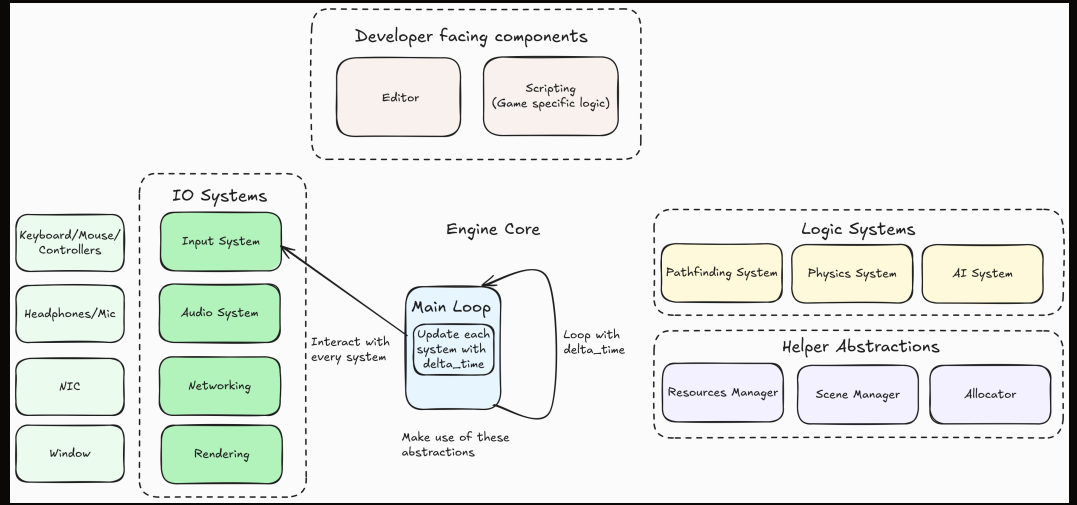
A game engine in 15 seconds

- **Main loop:** run a fixed set of stages every frame (or tick).
- **State:** world data (entities/components/resources/assets).
- **Scheduling:** order + parallelism + data dependencies. Automatic (advanced) or hard-coded in the engine.

```
1    while (running) {
2      poll_input();
3      run_systems(Update);       // gameplay, AI, scripting
4      run_systems(Physics);
5      run_systems(RenderPrep);   // culling, transforms, skinning
6      render();
7      present();
8    }
```

Here you can see a pseudocode of a main loop - our heart of the game engine that beats every tick. The systems update the state of the world, and it's done differently depending on the paradigm of which I will tell you later. The order of how it's done can be fixed or dynamic depending on the complexity of the engine. Here you can see it hardcoded.

Components of a game engine

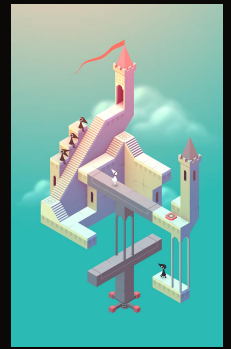


Credit: myself

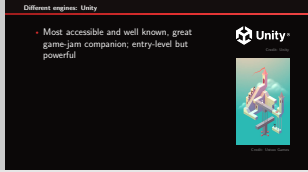
This picture shows more details, such as IO systems that directly interact with hardware, purely logical systems and some helper abstractions such as allocator or scene manager. There are also some developer-only components such as the game editor or scripting UI. Note that it is still a huge simplification where we assume that everything runs on a single tick.

- Most accessible and well known, great game-jam companion; entry-level but powerful

Credit: Unity



Credit: Ustwo Games

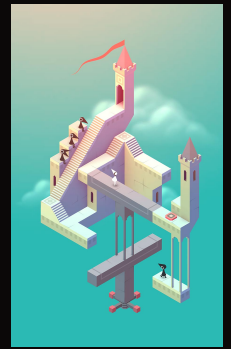


As our first example I picked my favourite non-OSS engine - Unity. It is a great starting point and for some it is even good-enough to create full-fledged games (albeit with some modifications and hacks to make it customized). It is object-oriented underneath with scripting via fixed API hooks. I highly recommend using it if you want quick iteration and a solid game-jam engine.

Different engines: Unity

- Most accessible and well known, great game-jam companion; entry-level but powerful
- Object-oriented programming via scripts, ECS (DOTS) is being rolled out

Credit: Unity



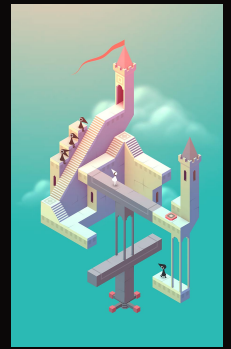
Credit: Ustwo Games

As our first example I picked my favourite non-OSS engine - Unity. It is a great starting point and for some it is even good-enough to create full-fledged games (albeit with some modifications and hacks to make it customized). It is object-oriented underneath with scripting via fixed API hooks. I highly recommend using it if you want quick iteration and a solid game-jam engine.

Different engines: Unity

- Most accessible and well known, great game-jam companion; entry-level but powerful
- Object-oriented programming via scripts, ECS (DOTS) is being rolled out
- Cannot modify the engine (closed source black box); instead - stable API with hooks

Credit: Unity



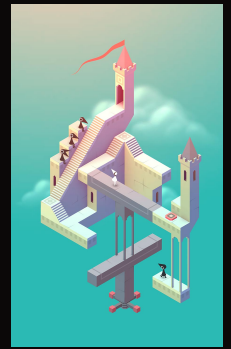
Credit: Ustwo Games

As our first example I picked my favourite non-OSS engine - Unity. It is a great starting point and for some it is even good-enough to create full-fledged games (albeit with some modifications and hacks to make it customized). It is object-oriented underneath with scripting via fixed API hooks. I highly recommend using it if you want quick iteration and a solid game-jam engine.

Different engines: Unity

- Most accessible and well known, great game-jam companion; entry-level but powerful
- Object-oriented programming via scripts, ECS (DOTS) is being rolled out
- Cannot modify the engine (closed source black box); instead - stable API with hooks
- Notable games: Monument Valley, Hollow Knight, Cuphead

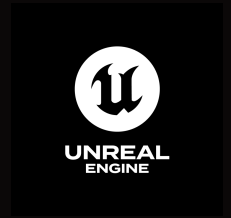
Credit: Unity



Credit: Ustwo Games

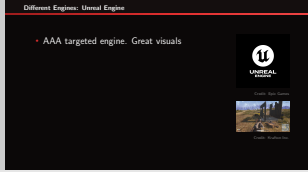
As our first example I picked my favourite non-OSS engine - Unity. It is a great starting point and for some it is even good-enough to create full-fledged games (albeit with some modifications and hacks to make it customized). It is object-oriented underneath with scripting via fixed API hooks. I highly recommend using it if you want quick iteration and a solid game-jam engine.

- AAA targeted engine. Great visuals



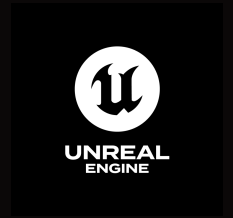
Credit: Epic Games

Credit: Krafton Inc.



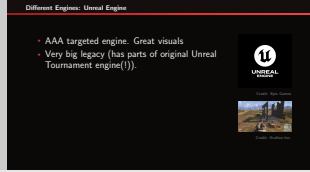
Most of you probably guessed the next big player. Unreal Engine is touted the AAA game engine and from the visual standpoint it really stands out. Be warned that it carries a ton of legacy code and is quite complex. One upside is that you don't need to be a programmer to use it because it allows for visual programming, although I personally did not like it)

- AAA targeted engine. Great visuals
- Very big legacy (has parts of original Unreal Tournament engine(!)).



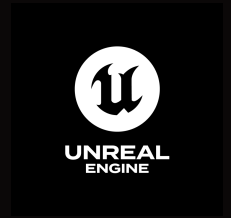
Credit: Epic Games

Credit: Krafton Inc.



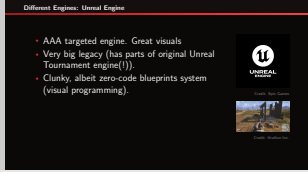
Most of you probably guessed the next big player. Unreal Engine is touted the AAA game engine and from the visual standpoint it really stands out. Be warned that it carries a ton of legacy code and is quite complex. One upside is that you don't need to be a programmer to use it because it allows for visual programming, although I personally did not like it)

- AAA targeted engine. Great visuals
- Very big legacy (has parts of original Unreal Tournament engine(!)).
- Clunky, albeit zero-code blueprints system (visual programming).



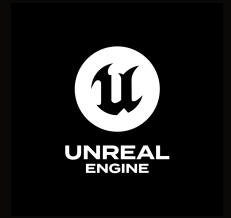
Credit: Epic Games

Credit: Krafton Inc.



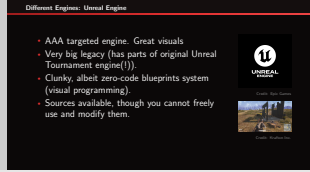
Most of you probably guessed the next big player. Unreal Engine is touted the AAA game engine and from the visual standpoint it really stands out. Be warned that it carries a ton of legacy code and is quite complex. One upside is that you don't need to be a programmer to use it because it allows for visual programming, although I personally did not like it)

- AAA targeted engine. Great visuals
- Very big legacy (has parts of original Unreal Tournament engine(!)).
- Clunky, albeit zero-code blueprints system (visual programming).
- Sources available, though you cannot freely use and modify them.



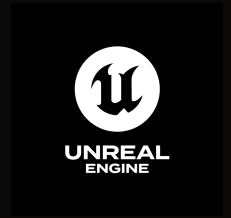
Credit: Epic Games

Credit: Krafton Inc.



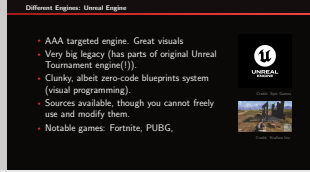
Most of you probably guessed the next big player. Unreal Engine is touted the AAA game engine and from the visual standpoint it really stands out. Be warned that it carries a ton of legacy code and is quite complex. One upside is that you don't need to be a programmer to use it because it allows for visual programming, although I personally did not like it)

- AAA targeted engine. Great visuals
- Very big legacy (has parts of original Unreal Tournament engine(!)).
- Clunky, albeit zero-code blueprints system (visual programming).
- Sources available, though you cannot freely use and modify them.
- Notable games: Fortnite, PUBG,



Credit: Epic Games

Credit: Krafton Inc.

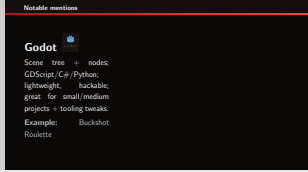


Most of you probably guessed the next big player. Unreal Engine is touted the AAA game engine and from the visual standpoint it really stands out. Be warned that it carries a ton of legacy code and is quite complex. One upside is that you don't need to be a programmer to use it because it allows for visual programming, although I personally did not like it)

Godot

Scene tree + nodes; GDScript/C#/Python; lightweight, hackable; great for small/medium projects + tooling tweaks.

Example: Buckshot Roulette



There are a lot of game engines and I chose just a few of them. The next popular and open source engine is Godot, which recently gained a lot of new followers as people turned their backs on Unity and Unreal due to disputes over royalties from published games to these engine owning companies. Next up is O3DE which is developed under the Linux Foundation umbrella and is somewhat a successor to CryEngine which you all probably know from the visually stunning Crysis and Far Cry games. Last we have Bevy, which is the most widely used Rust engine. It is a pioneering effort showing that Rust is suitable for gamedev as well.

Godot

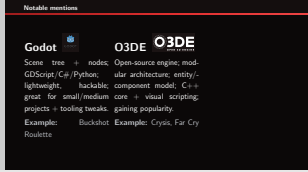
Scene tree + nodes; GDScript/C#/Python; lightweight, hackable; great for small/medium projects + tooling tweaks.

Example: Buckshot Roulette

O3DE

Open-source engine; modular architecture; entity/-component model; C++ core + visual scripting; gaining popularity.

Example: Crysis, Far Cry



There are a lot of game engines and I chose just a few of them. The next popular and open source engine is Godot, which recently gained a lot of new followers as people turned their backs on Unity and Unreal due to disputes over royalties from published games to these engine owning companies. Next up is O3DE which is developed under the Linux Foundation umbrella and is somewhat a successor to CryEngine which you all probably know from the visually stunning Crysis and Far Cry games. Last we have Bevy, which is the most widely used Rust engine. It is a pioneering effort showing that Rust is suitable for gamedev as well.

Godot

Scene tree + nodes; GDScript/C#/Python; lightweight, hackable; great for small/medium projects + tooling tweaks.

Example: Buckshot Roulette

O3DE

Open-source engine; modular architecture; entity/-component model; C++ core + visual scripting; gaining popularity.

Example: Crysis, Far Cry

Bevy

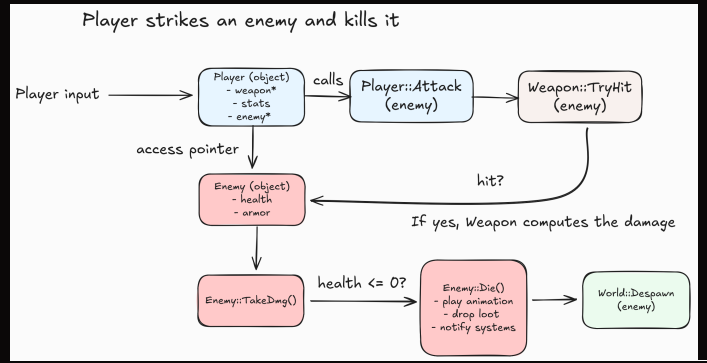
Pure ECS in Rust; data-driven systems/schedules; code-first; editor/tooling still maturing; big Rust gamedev pioneer.

Example: TinyGlade

There are a lot of game engines and I chose just a few of them. The next popular and open source engine is Godot, which recently gained a lot of new followers as people turned their backs on Unity and Unreal due to disputes over royalties from published games to these engine owning companies. Next up is O3DE which is developed under the Linux Foundation umbrella and is somewhat a successor to CryEngine which you all probably know from the visually stunning Crysis and Far Cry games. Last we have Bevy, which is the most widely used Rust engine. It is a pioneering effort showing that Rust is suitable for gamedev as well.

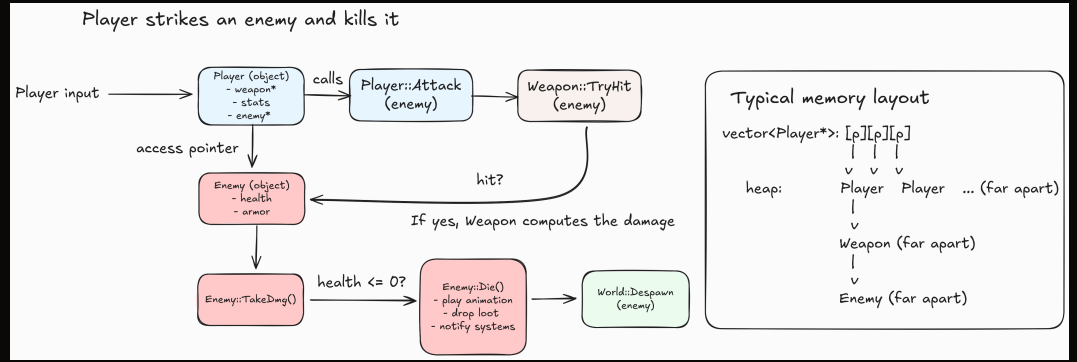
What is OOP?

Object-Oriented Programming approach

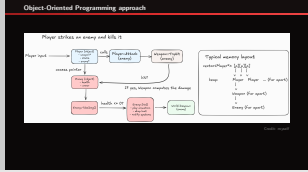


Credit: myself

In the Object Oriented paradigm we act upon objects that are akin how we interact with reality around us. So when a player hits an enemy, it acts on the the pointer to the enemy object. In turn asking the enemy to deduct the amount of damage taken from health field stored in this object. If the enemy dies, it has to be despawned and this is one of the weak points of this paradigm.

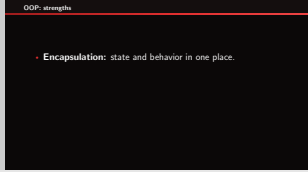


Credit: myself



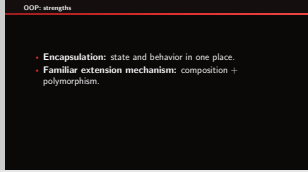
As you can see on the right side, we have a long pointer chasing chain, and I hope you already know that it is something that is not the best for our performance because first of all the pointers may be null and secondly we jump around the heap, thrashing the cache lines. This is why large games often stop scaling, and why performance-heavy parts move away from object graphs."

- **Encapsulation:** state and behavior in one place.



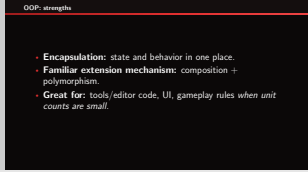
The reason why OOP is so popular is because it feels natural for us humans to reason about. We can create objects that interact with each other just as they do in real life. Want a character to shoot an enemy - just add the shoot function on top of the character. No boilerplate, yet we pay a price for that. It is still widely used in places like tools or editors. If you don't need heaps of performance you can code pretty complex games in it as well.

- **Encapsulation:** state and behavior in one place.
- **Familiar extension mechanism:** composition + polymorphism.



The reason why OOP is so popular is because it feels natural for us humans to reason about. We can create objects that interact with each other just as they do in real life. Want a character to shoot an enemy - just add the shoot function on top of the character. No boilerplate, yet we pay a price for that. It is still widely used in places like tools or editors. If you don't need heaps of performance you can code pretty complex games in it as well.

- **Encapsulation:** state and behavior in one place.
- **Familiar extension mechanism:** composition + polymorphism.
- **Great for:** tools/editor code, UI, gameplay rules *when unit counts are small*.



The reason why OOP is so popular is because it feels natural for us humans to reason about. We can create objects that interact with each other just as they do in real life. Want a character to shoot an enemy - just add the shoot function on top of the character. No boilerplate, yet we pay a price for that. It is still widely used in places like tools or editors. If you don't need heaps of performance you can code pretty complex games in it as well.

- **Memory:** objects spread across heap ⇒ cache misses, TLB misses, allocator fragmentation.

OOP: typical bottlenecks
- **Memory:** objects spread across heap ⇒ cache misses, TLB misses, allocator fragmentation.

The drawbacks mentioned here are the main reasons why we should consider different approaches. With that many objects we are fragmenting our heap and make our CPU's job much harder when it comes to any predictions which results in cache thrashing. What is simple for humans is not necessarily as simple for computers. Not having data nicely packed in memory makes it much harder to use SIMD.

- **Memory:** objects spread across heap ⇒ cache misses, TLB misses, allocator fragmentation.
- **Data access:** pointer chasing (address-dependent loads) ⇒ latency dominates.

OOP: typical bottlenecks
- **Memory:** objects spread across heap ⇒ cache misses, TLB misses, allocator fragmentation.
- **Data access:** pointer chasing (address-dependent loads) ⇒ latency dominates.

The drawbacks mentioned here are the main reasons why we should consider different approaches. With that many objects we are fragmenting our heap and make our CPU's job much harder when it comes to any predictions which results in cache thrashing. What is simple for humans is not necessarily as simple for computers. Not having data nicely packed in memory makes it much harder to use SIMD.

- **Memory:** objects spread across heap ⇒ cache misses, TLB misses, allocator fragmentation.
- **Data access:** pointer chasing (address-dependent loads) ⇒ latency dominates.
- **Control flow:** indirect/virtual calls + many branches ⇒ mispredicts, hard to batch/SIMD.

OOP: typical bottlenecks
- **Memory:** objects spread across heap ⇒ cache misses, TLB misses, allocator fragmentation.
- **Data access:** pointer chasing (address-dependent loads) ⇒ latency dominates.
- **Control flow:** indirect/virtual calls + many branches ⇒ mispredicts, hard to batch/SIMD.

The drawbacks mentioned here are the main reasons why we should consider different approaches. With that many objects we are fragmenting our heap and make our CPU's job much harder when it comes to any predictions which results in cache thrashing. What is simple for humans is not necessarily as simple for computers. Not having data nicely packed in memory makes it much harder to use SIMD.

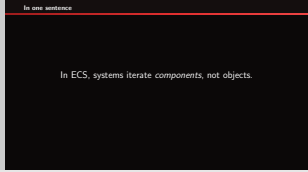
- **Memory:** objects spread across heap ⇒ cache misses, TLB misses, allocator fragmentation.
- **Data access:** pointer chasing (address-dependent loads) ⇒ latency dominates.
- **Control flow:** indirect/virtual calls + many branches ⇒ mispredicts, hard to batch/SIMD.
- **Event cascades:** e.g. Death triggers many callbacks ⇒ unpredictable fan-out and more misses.

OOP: typical bottlenecks
- Memory: objects spread across heap ⇒ cache misses, TLB misses, allocator fragmentation.
- Data access: pointer chasing (address-dependent loads) ⇒ latency dominates.
- Control flow: indirect/virtual calls + many branches ⇒ mispredicts, hard to batch/SIMD.
- Event cascades: e.g. Death triggers many callbacks ⇒ unpredictable fan-out and more misses.

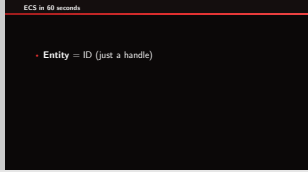
The drawbacks mentioned here are the main reasons why we should consider different approaches. With that many objects we are fragmenting our heap and make our CPU's job much harder when it comes to any predictions which results in cache thrashing. What is simple for humans is not necessarily as simple for computers. Not having data nicely packed in memory makes it much harder to use SIMD.

What is ECS?

In ECS, systems iterate *components*, not objects.

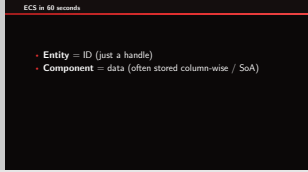


- **Entity** = ID (just a handle)



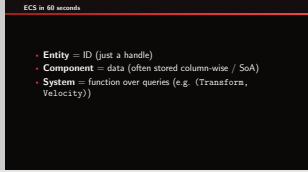
This is where Entity Component System comes to the rescue. It is a Data-Oriented approach where we focus on performance instead of straightforwardness. In ECS we have systems iterating over components that map to particular entities. So querying all entities containing particular components becomes much easier.

- **Entity** = ID (just a handle)
- **Component** = data (often stored column-wise / SoA)



This is where Entity Component System comes to the rescue. It is a Data-Oriented approach where we focus on performance instead of straightforwardness. In ECS we have systems iterating over components that map to particular entities. So querying all entities containing particular components becomes much easier.

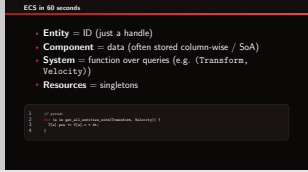
- **Entity** = ID (just a handle)
- **Component** = data (often stored column-wise / SoA)
- **System** = function over queries (e.g. `(Transform, Velocity)`)



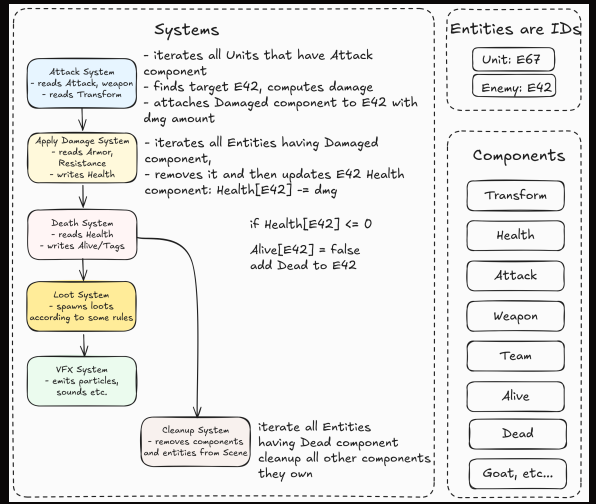
This is where Entity Component System comes to the rescue. It is a Data-Oriented approach where we focus on performance instead of straightforwardness. In ECS we have systems iterating over components that map to particular entities. So querying all entities containing particular components becomes much easier.

- **Entity** = ID (just a handle)
- **Component** = data (often stored column-wise / SoA)
- **System** = function over queries (e.g. `(Transform, Velocity)`)
- **Resources** = singletons

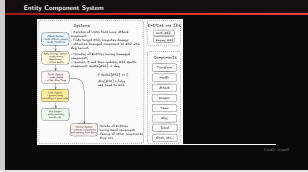
```
// pseudo
for (e in get_all_entities_with(Transform, Velocity)) {
  T[e].pos += V[e].v * dt;
}
```

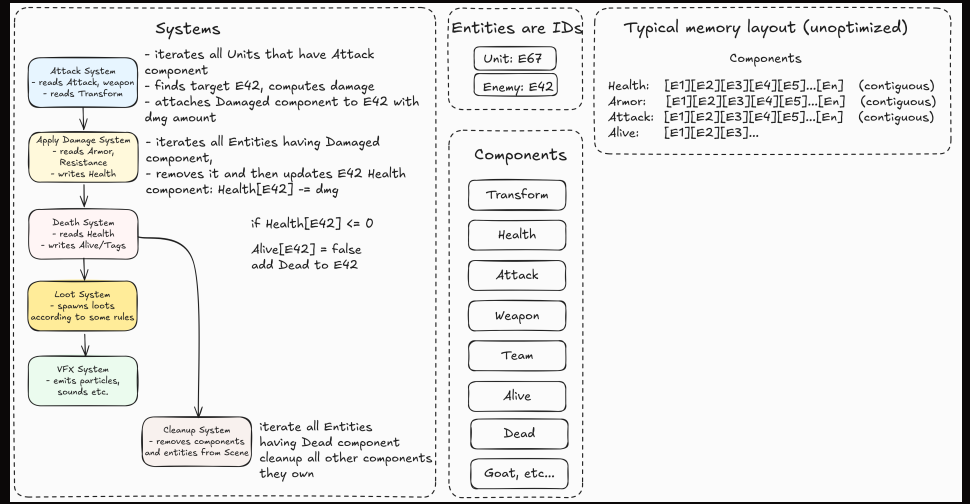
This is where Entity Component System comes to the rescue. It is a Data-Oriented approach where we focus on performance instead of straightforwardness. In ECS we have systems iterating over components that map to particular entities. So querying all entities containing particular components becomes much easier.



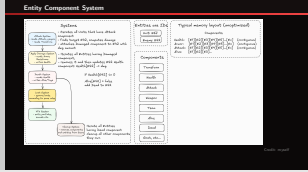
Credit: myself



Here we have the same scenario as before, where a player strikes and fells an enemy. In this case we will sequentially step through all the systems starting with Attack System. We iterate all units that have the Attack component, we find our target and compute how much damage should be done to it. We attach a Damaged component with amount of damage received. All actions of damaging other units are done during next iterations of this system. The we have the system responsible for applying the damage that will subtract the damage received from Unit's total health and if it's supposed to be dead it will add a Dead component that is read by the Cleanup System at the very end. We also have auxiliary systems responsible for spawning loot and emitting some particles. Usually the last systems to run are those that modify the world's state so Spawn/Cleanup ones.

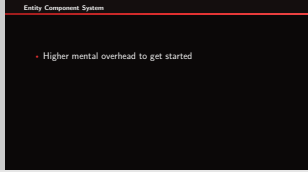


Credit: myself



Contrasted to OOP, we can clearly see we have arrays of components that contain indices of entities (or some other lightweight markers) that own them. This makes iterating over sets of components more cache-friendly.

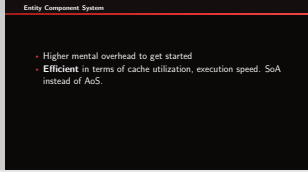
- Higher mental overhead to get started



We tradeoff ease of understanding for performance as we have all components laying next to each other in memory.

The drawback is that it is slightly more difficult to debug when something goes wrong.

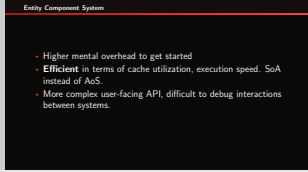
- Higher mental overhead to get started
- **Efficient** in terms of cache utilization, execution speed. SoA instead of AoS.



We tradeoff ease of understanding for performance as we have all components laying next to each other in memory.

The drawback is that it is slightly more difficult to debug when something goes wrong.

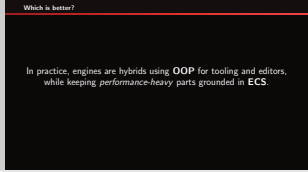
- Higher mental overhead to get started
- **Efficient** in terms of cache utilization, execution speed. SoA instead of AoS.
- More complex user-facing API, difficult to debug interactions between systems.



We tradeoff ease of understanding for performance as we have all components laying next to each other in memory.

The drawback is that it is slightly more difficult to debug when something goes wrong.

In practice, engines are hybrids using **OOP** for tooling and editors, while keeping *performance-heavy* parts grounded in **ECS**.



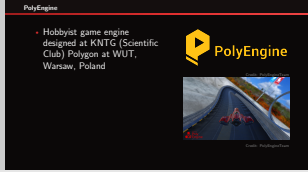
What is PolyEngine?

- Hobbyist game engine designed at KNTG (Scientific Club) Polygon at WUT, Warsaw, Poland

PolyEngine

Credit: PolyEngineTeam

Credit: PolyEngineTeam



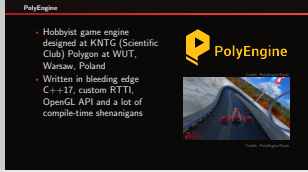
Our first case study is PolyEngine, a project which taught me proper modern C++ and some of the best gamedev patterns. Developed by a few talented members of Polygon, it was inspired by Overwatch's novel ECS architecture.

- Hobbyist game engine designed at KNTG (Scientific Club) Polygon at WUT, Warsaw, Poland
- Written in bleeding edge C++17, custom RTTI, OpenGL API and a lot of compile-time shenanigans

PolyEngine

Credit: PolyEngineTeam

Credit: PolyEngineTeam



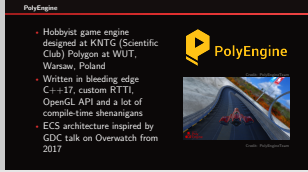
Our first case study is PolyEngine, a project which taught me proper modern C++ and some of the best gamedev patterns. Developed by a few talented members of Polygon, it was inspired by Overwatch's novel ECS architecture.

- Hobbyist game engine designed at KNTG (Scientific Club) Polygon at WUT, Warsaw, Poland
- Written in bleeding edge C++17, custom RTTI, OpenGL API and a lot of compile-time shenanigans
- ECS architecture inspired by GDC talk on Overwatch from 2017

PolyEngine

Credit: PolyEngineTeam

Credit: PolyEngineTeam



Our first case study is PolyEngine, a project which taught me proper modern C++ and some of the best gamedev patterns. Developed by a few talented members of Polygon, it was inspired by Overwatch's novel ECS architecture.

PolyEngine: RTTI

- Custom RTTI is not just `typeid`: it describes **properties** (offset/type/flags).

PolyEngine: RTTI
- Custom RTTI is not just typeid: it describes properties (offset/type/flags).

Here we can see an example of how the Run Time Type Inference works. It is based on a couple of macros that do all the internal magic. Adding those allows for serialization and deserialization of scene as well as seamless plugging into Qt.

PolyEngine: RTTI

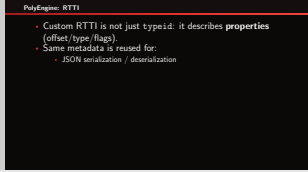
- Custom RTTI is not just `typeid`: it describes **properties** (offset/type/flags).
- Same metadata is reused for:

PolyEngine: RTTI
- Custom RTTI is not just typeid: it describes properties (offset/type/flags).
- Same metadata is reused for:

Here we can see an example of how the Run Time Type Inference works. It is based on a couple of macros that do all the internal magic. Adding those allows for serialization and deserialization of scene as well as seamless plugging into Qt.

PolyEngine: RTTI

- Custom RTTI is not just `typeid`: it describes **properties** (offset/type/flags).
- Same metadata is reused for:
  - JSON serialization / deserialization

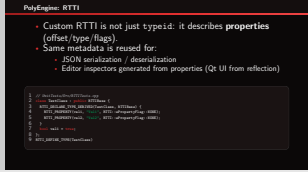


Here we can see an example of how the Run Time Type Inference works. It is based on a couple of macros that do all the internal magic. Adding those allows for serialization and deserialization of scene as well as seamless plugging into Qt.

PolyEngine: RTTI

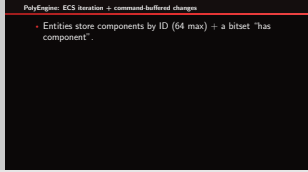
- Custom RTTI is not just `typeid`: it describes **properties** (offset/type/flags).
- Same metadata is reused for:
  - JSON serialization / deserialization
  - Editor inspectors generated from properties (Qt UI from reflection)

```cpp
// UnitTests/Src/RTTITests.cpp
class TestClass : public RTTIBase {
  RTTI_DECLARE_TYPE_DERIVED(TestClass, RTTIBase) {
    RTTI_PROPERTY(val1, "Val1", RTTI::ePropertyFlag::NONE);
    RTTI_PROPERTY(val2, "Val2", RTTI::ePropertyFlag::NONE);
  }
  bool val1 = true;
};
RTTI_DEFINE_TYPE(TestClass)
```

Here we can see an example of how the Run Time Type Inference works. It is based on a couple of macros that do all the internal magic. Adding those allows for serialization and deserialization of scene as well as seamless plugging into Qt.

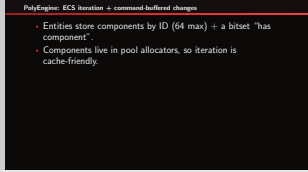
- Entities store components by ID (64 max) + a bitset “has component”.



Another cool feature is being able to iterate the components using C++17 structure binding. Moreover, the iteration is over memory-pools so we preserve proper memory layouts without fragmentation. Last noteworthy feature is having the deferred system that acts as a command buffer for adding/removing entities or perfoming other major changes on the game world.

PolyEngine: ECS iteration + command-buffered changes

- Entities store components by ID (64 max) + a bitset “has component”.
- Components live in pool allocators, so iteration is cache-friendly.

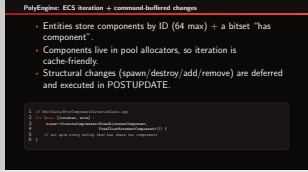


Another cool feature is being able to iterate the components using C++17 structure binding. Moreover, the iteration is over memory-pools so we preserve proper memory layouts without fragmentation. Last noteworthy feature is having the deferred system that acts as a command buffer for adding/removing entities or perfoming other major changes on the game world.

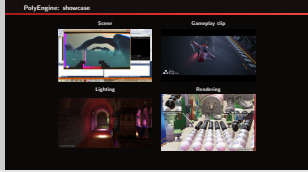
PolyEngine: ECS iteration + command-buffered changes

- Entities store components by ID (64 max) + a bitset “has component”.
- Components live in pool allocators, so iteration is cache-friendly.
- Structural changes (spawn/destroy/add/remove) are deferred and executed in POSTUPDATE.

```cpp
// UnitTests/Src/ComponentIterationTests.cpp
for (auto [listener, move] :
     scene->IterateComponents<SoundListenerComponent,
                              FreeFloatMovementComponent>()) {
    // act upon every entity that has these two components
}
```

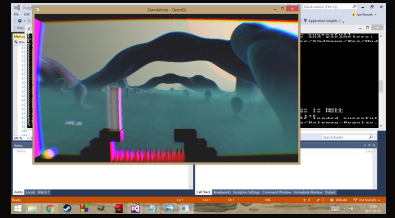



Another cool feature is being able to iterate the components using C++17 structure binding. Moreover, the iteration is over memory-pools so we preserve proper memory layouts without fragmentation. Last noteworthy feature is having the deferred system that acts as a command buffer for adding/removing entities or perfoming other major changes on the game world.



PolyEngine: showcase

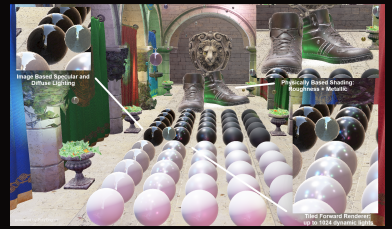
Scene



Gameplay clip

Lighting

Rendering



Here you can see a few snippets from the engine. First is a screenshot from a game developed during one of our gamejams. Then we have a gameplay clip. In the bottom row we have demos of advanced lighting.

- **Plugin architecture:** Standalone loads `libRenderingDevice` + `libGame` at runtime (fast iteration, swappable backends).

PolyEngine: what went right
- **Plugin architecture:** Standalone loads `libRenderingDevice` + `libGame` at runtime (fast iteration, swappable backends).

Though the project is now discontinued, it shines in a lot of aspects, such as modern architecture allowing for hot-reloading the game. Having custom reflection which allowed for writing an editor in Qt without much overhead. A good ECS - approach, ambitious renderer and most importantly a ton of Unit tests.

- **Plugin architecture:** Standalone loads `libRenderingDevice` + `libGame` at runtime (fast iteration, swappable backends).
- **RTTI:** custom reflection + property metadata → JSON serialization + editor inspectors.

PolyEngine: what went right
- **Plugin architecture:** Standalone loads `libRenderingDevice` + `libGame` at runtime (fast iteration, swappable backends).
- **RTTI:** custom reflection + property metadata → JSON serialization + editor inspectors.

Though the project is now discontinued, it shines in a lot of aspects, such as modern architecture allowing for hot-reloading the game. Having custom reflection which allowed for writing an editor in Qt without much overhead. A good ECS - approach, ambitious renderer and most importantly a ton of Unit tests.

- **Plugin architecture:** Standalone loads `libRenderingDevice` + `libGame` at runtime (fast iteration, swappable backends).
- **RTTI:** custom reflection + property metadata → JSON serialization + editor inspectors.
- **ECS:** pool allocators + bitset "has component" + C++17-friendly query API.

PolyEngine: what went right
- **Plugin architecture:** Standalone loads `libRenderingDevice` + `libGame` at runtime (fast iteration, swappable backends).
- **RTTI:** custom reflection + property metadata → JSON serialization + editor inspectors.
- **ECS:** pool allocators + bitset "has component" + C++17-friendly query API.

Though the project is now discontinued, it shines in a lot of aspects, such as modern architecture allowing for hot-reloading the game. Having custom reflection which allowed for writing an editor in Qt without much overhead. A good ECS - approach, ambitious renderer and most importantly a ton of Unit tests.

- **Deferred pattern:** deferred task queue (command buffer) executed in a late phase.

PolyEngine: what went right
- **Deferred pattern:** deferred task queue (command buffer) executed in a late phase.

- **Deferred pattern:** deferred task queue (command buffer) executed in a late phase.
- **Rendering:** tiled forward renderer + postFX pipeline in OpenGL.

PolyEngine: what went right
- **Deferred pattern:** deferred task queue (command buffer) executed in a late phase.
- **Rendering:** tiled forward renderer + postFX pipeline in OpenGL.

- **Deferred pattern:** deferred task queue (command buffer) executed in a late phase.
- **Rendering:** tiled forward renderer + postFX pipeline in OpenGL.
- **Tests:** unit tests saved countless hours of refactors.

PolyEngine: what went right
- **Deferred pattern:** deferred task queue (command buffer) executed in a late phase.
- **Rendering:** tiled forward renderer + postFX pipeline in OpenGL.
- **Tests:** unit tests saved countless hours of refactors.

- **Not benchmarking early enough kills the performance:** it was too late once we realized that.

PolyEngine: lessons learned
- Not benchmarking early enough kills the performance: it was too late once we realized that.

Measure early, optimize late — and only what the profiler points to. As mentioned earlier, the project is now over. But the legacy lives on! Feel free to take heed of some of our lessons. What was probably the biggest showstopper is that we didn't benchmark the core systems often and early. Focusing instead on minor premature optimizations like writing our custom Priority Queue where we could have used the std-provided one. Also, next time I would not use Qt for my editor, rather choosing something higher-level.

- **Not benchmarking early enough kills the performance:** it was too late once we realized that.
- **Writing an editor in C++ is overkill:** Qt is elegant, but development speed is abysmal.

PolyEngine: lessons learned
- Not benchmarking early enough kills the performance: it was too late once we realized that.
- Writing an editor in C++ is overkill: Qt is elegant, but development speed is abysmal.

Measure early, optimize late — and only what the profiler points to. As mentioned earlier, the project is now over. But the legacy lives on! Feel free to take heed of some of our lessons. What was probably the biggest showstopper is that we didn't benchmark the core systems often and early. Focusing instead on minor premature optimizations like writing our custom Priority Queue where we could have used the std-provided one. Also, next time I would not use Qt for my editor, rather choosing something higher-level.

- **Not benchmarking early enough kills the performance:** it was too late once we realized that.
- **Writing an editor in C++ is overkill:** Qt is elegant, but development speed is abysmal.
- **Premature optimization is the root of all evil:** We wrote custom containers, like hashmap. Turns out it's hard to outdo STL.

PolyEngine: lessons learned
- **Not benchmarking early enough kills the performance:** it was too late once we realized that.
- **Writing an editor in C++ is overkill:** Qt is elegant, but development speed is abysmal.
- **Premature optimization is the root of all evil:** We wrote custom containers, like hashmap. Turns out it's hard to outdo STL.

Measure early, optimize late — and only what the profiler points to. As mentioned earlier, the project is now over. But the legacy lives on! Feel free to take heed of some of our lessons. What was probably the biggest showstopper is that we didn't benchmark the core systems often and early. Focusing instead on minor premature optimizations like writing our custom Priority Queue where we could have used the std-provided one. Also, next time I would not use Qt for my editor, rather choosing something higher-level.

- **Not benchmarking early enough kills the performance:** it was too late once we realized that.
- **Writing an editor in C++ is overkill:** Qt is elegant, but development speed is abysmal.
- **Premature optimization is the root of all evil:** We wrote custom containers, like hashmap. Turns out it's hard to outdo STL.
- **Rendering pipeline too heavy:** OpenGL versus more modern Vulkan.

PolyEngine: lessons learned
- **Not benchmarking early enough kills the performance:** it was too late once we realized that.
- **Writing an editor in C++ is overkill:** Qt is elegant, but development speed is abysmal.
- **Premature optimization is the root of all evil:** We wrote custom containers, like hashmap. Turns out it's hard to outdo STL.
- **Rendering pipeline too heavy:** OpenGL versus more modern Vulkan.

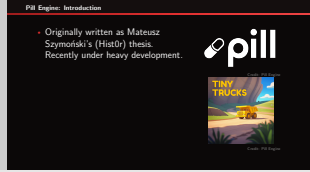
Measure early, optimize late — and only what the profiler points to. As mentioned earlier, the project is now over. But the legacy lives on! Feel free to take heed of some of our lessons. What was probably the biggest showstopper is that we didn't benchmark the core systems often and early. Focusing instead on minor premature optimizations like writing our custom Priority Queue where we could have used the std-provided one. Also, next time I would not use Qt for my editor, rather choosing something higher-level.

What is Pill Engine?

- Originally written as Mateusz Szymoński's (Hist0r) thesis. Recently under heavy development.

Credit: Pill Engine

Credit: Pill Engine

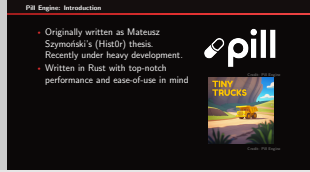


We come to our almost last section and spiritual successor to PolyEngine - Pill Engine. This was initially a bachelor's thesis effort that turned out to be solid enough to build a proper engine out of it. It is written in Rust instead of C++ and with performance and modern rendering capabilities in mind.

- Originally written as Mateusz Szymoński's (Hist0r) thesis. Recently under heavy development.
- Written in Rust with top-notch performance and ease-of-use in mind

Credit: Pill Engine

Credit: Pill Engine

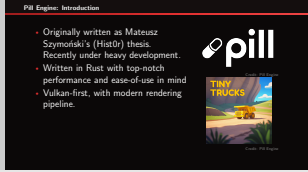


We come to our almost last section and spiritual successor to PolyEngine - Pill Engine. This was initially a bachelor's thesis effort that turned out to be solid enough to build a proper engine out of it. It is written in Rust instead of C++ and with performance and modern rendering capabilities in mind.

- Originally written as Mateusz Szymoński's (Hist0r) thesis. Recently under heavy development.
- Written in Rust with top-notch performance and ease-of-use in mind
- Vulkan-first, with modern rendering pipeline.

Credit: Pill Engine

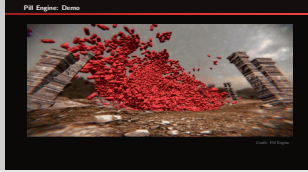
Credit: Pill Engine



We come to our almost last section and spiritual successor to PolyEngine - Pill Engine. This was initially a bachelor's thesis effort that turned out to be solid enough to build a proper engine out of it. It is written in Rust instead of C++ and with performance and modern rendering capabilities in mind.

Pill Engine: Demo

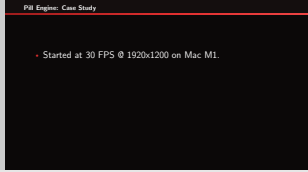
Credit: Pill Engine



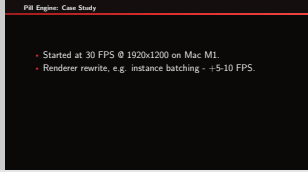
How *fast* can we render 60 000 pills?

Pill Engine: Case Study
How *fast* can we render 60 000 pills?

- Started at 30 FPS @ 1920x1200 on Mac M1.



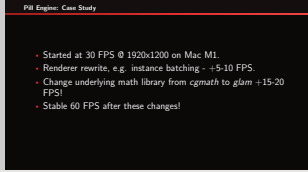
- Started at 30 FPS @ 1920x1200 on Mac M1.
- Renderer rewrite, e.g. instance batching - +5-10 FPS.



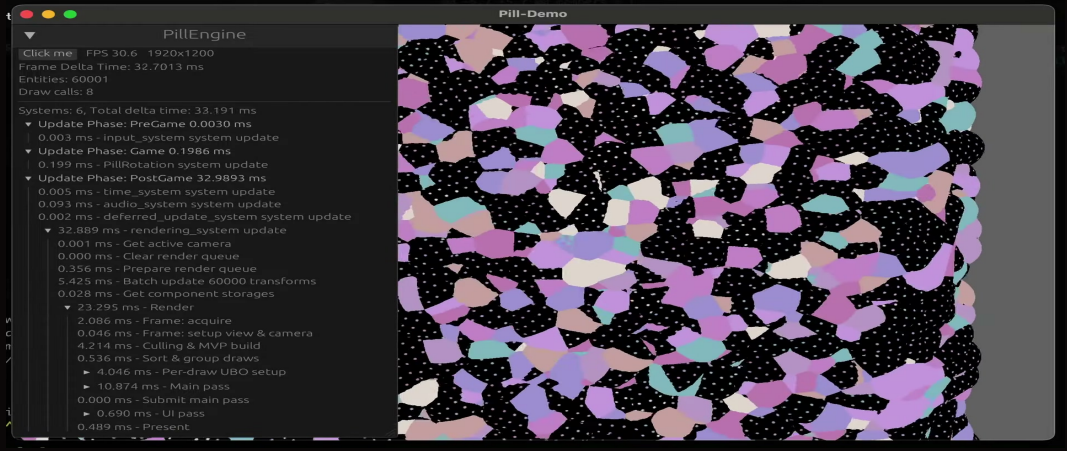
- Started at 30 FPS @ 1920x1200 on Mac M1.
- Renderer rewrite, e.g. instance batching - +5-10 FPS.
- Change underlying math library from *cgmath* to *glam* +15-20 FPS!

Pill Engine: Case Study
- Started at 30 FPS @ 1920x1200 on Mac M1.
- Renderer rewrite, e.g. instance batching - +5-10 FPS.
- Change underlying math library from *cgmath* to *glam* +15-20 FPS!

- Started at 30 FPS @ 1920x1200 on Mac M1.
- Renderer rewrite, e.g. instance batching - +5-10 FPS.
- Change underlying math library from *cgmath* to *glam* +15-20 FPS!
- Stable 60 FPS after these changes!

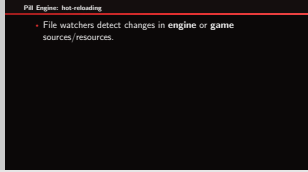


Pill Engine: Case Study



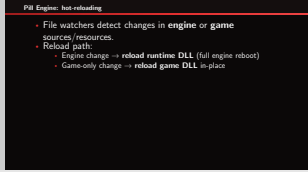
Credit: sp0lsh (Michał Kłoś)

- File watchers detect changes in **engine** or **game** sources/resources.



This is “real hot reload” in two layers. Engine edits rebuild and reload the runtime DLL. Game edits just swap the game DLL and call reload_game() through a stable ABI.

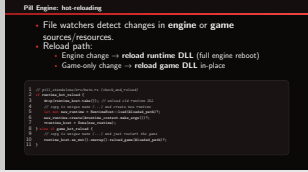
- File watchers detect changes in **engine** or **game** sources/resources.
- Reload path:
  - Engine change → **reload runtime DLL** (full engine reboot)
  - Game-only change → **reload game DLL** in-place



This is “real hot reload” in two layers. Engine edits rebuild and reload the runtime DLL. Game edits just swap the game DLL and call reload_game() through a stable ABI.

- File watchers detect changes in **engine** or **game** sources/resources.
- Reload path:
  - Engine change → **reload runtime DLL** (full engine reboot)
  - Game-only change → **reload game DLL** in-place

```rust
// pill_standalone/src/main.rs (check_and_reload)
if runtime_hot_reload {
    drop(runtime_host.take()); // unload old runtime DLL
    // copy to unique name (...) and create new runtime
    let mut new_runtime = RuntimeHost::load(&loaded_path)?;
    new_runtime.create(&runtime_context.make_args())?;
    *runtime_host = Some(new_runtime);
} else if game_hot_reload {
    // copy to unique name (...) and just restart the game
    runtime_host.as_mut().unwrap().reload_game(&loaded_path)?;
}
```

This is “real hot reload” in two layers. Engine edits rebuild and reload the runtime DLL. Game edits just swap the game DLL and call reload_game() through a stable ABI.

Pill Engine: TypeId mismatch issue

- Hot-reload with dylibs can build **two copies** of shared crates breaking `TypeId`, downcasting, and trait-object assumptions across DLL boundaries.

Pill Engine: TypeId mismatch issue
- Hot-reload with dylibs can build **two copies** of shared crates breaking `TypeId`, downcasting, and trait-object assumptions across DLL boundaries.

This is the "Rust hot-reload tax": two crate graphs = two TypeIds. The launcher fixes it deterministically by linking the game crate into the engine workspace and forcing a shared build graph.

Pill Engine: TypeId mismatch issue

- Hot-reload with dylibs can build **two copies** of shared crates breaking `TypeId`, downcasting, and trait-object assumptions across DLL boundaries.
- PillLauncher forces the game to compile inside the **engine workspace**.

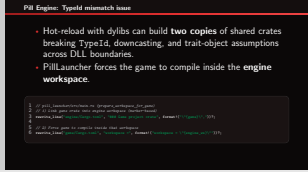
Pill Engine: TypeId mismatch issue
- Hot-reload with dylibs can build **two copies** of shared crates breaking `TypeId`, downcasting, and trait-object assumptions across DLL boundaries.
- PillLauncher forces the game to compile inside the **engine workspace**.

This is the "Rust hot-reload tax": two crate graphs = two TypeIds. The launcher fixes it deterministically by linking the game crate into the engine workspace and forcing a shared build graph.

- Hot-reload with dylibs can build **two copies** of shared crates breaking `TypeId`, downcasting, and trait-object assumptions across DLL boundaries.
- PillLauncher forces the game to compile inside the **engine workspace**.

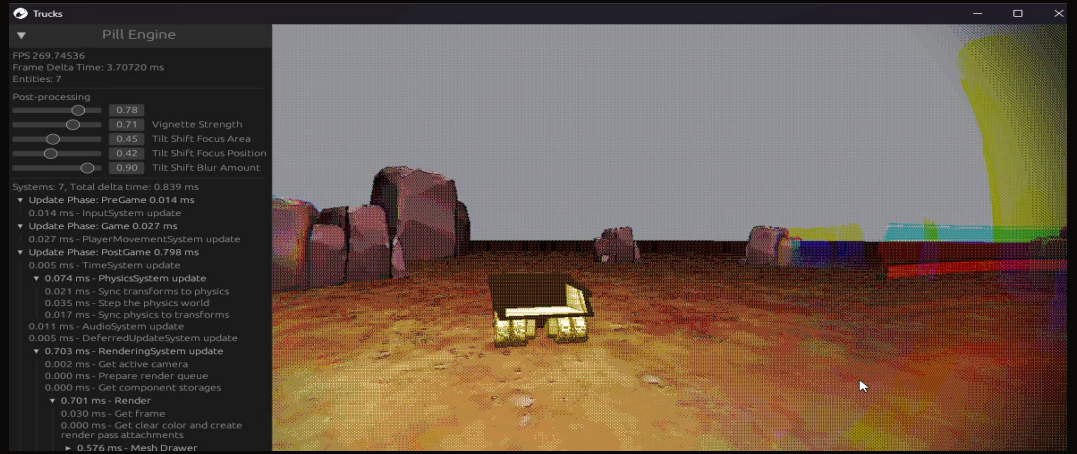
```rust
// pill_launcher/src/main.rs (prepare_workspace_for_game)
// 1) Link game crate into engine workspace (marker-based)
rewrite_line("engine/Cargo.toml", "### Game project crate", format!("\"{game}\",",))?;

// 2) Force game to compile inside that workspace
rewrite_line("game/Cargo.toml", "workspace =", format!("workspace = \"{engine_ws}\""))?;
```

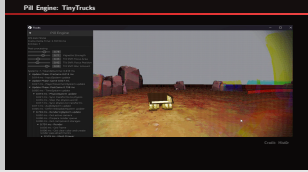



This is the "Rust hot-reload tax": two crate graphs = two TypeIds. The launcher fixes it deterministically by linking the game crate into the engine workspace and forcing a shared build graph.

Pill Engine: TinyTrucks



Credit: Hist0r



- **Stateful hot reload:** 🔧 serialize world/resources → reload game DLL → restore state.

Pill Engine: Roadmap
- Stateful hot reload: 🔧 serialize world/resources → reload game DLL → restore state.

The future looks bright as we have a lot of changes underway. Engine and game hot-reloading with state preservation aim make developers' experience way better. The next big thing is changing the ECS to use archetypes which is a big speedup over current way of doing things. We also plan to allow for having personalized graphs of system dependencies. We want for the engine to be runnable in browser and have a fast and intuitive editor. Lastly, we will strive to optimize the renderer and engine to support mobiles.

- **Stateful hot reload:** 🔧 serialize world/resources → reload game DLL → restore state.
- **Archetype-based ECS:** 🔧 extension of the ECS to support archetype-based iteration.

Pill Engine: Roadmap
- **Stateful hot reload:** 🔧 serialize world/resources → reload game DLL → restore state.
- **Archetype-based ECS:** 🔧 extension of the ECS to support archetype-based iteration.

The future looks bright as we have a lot of changes underway. Engine and game hot-reloading with state preservation aim make developers' experience way better. The next big thing is changing the ECS to use archetypes which is a big speedup over current way of doing things. We also plan to allow for having personalized graphs of system dependencies. We want for the engine to be runnable in browser and have a fast and intuitive editor. Lastly, we will strive to optimize the renderer and engine to support mobiles.

- **Stateful hot reload:** 🔧 serialize world/resources → reload game DLL → restore state.
- **Archetype-based ECS:** 🔧 extension of the ECS to support archetype-based iteration.
- **Scheduling:** allow for specifying dependencies between systems (add parallelism).

Pill Engine: Roadmap
- **Stateful hot reload:** 🔧 serialize world/resources → reload game DLL → restore state.
- **Archetype-based ECS:** 🔧 extension of the ECS to support archetype-based iteration.
- **Scheduling:** allow for specifying dependencies between systems (add parallelism).

The future looks bright as we have a lot of changes underway. Engine and game hot-reloading with state preservation aim make developers' experience way better. The next big thing is changing the ECS to use archetypes which is a big speedup over current way of doing things. We also plan to allow for having personalized graphs of system dependencies. We want for the engine to be runnable in browser and have a fast and intuitive editor. Lastly, we will strive to optimize the renderer and engine to support mobiles.

- **Stateful hot reload:** 🔧 serialize world/resources → reload game DLL → restore state.
- **Archetype-based ECS:** 🔧 extension of the ECS to support archetype-based iteration.
- **Scheduling:** allow for specifying dependencies between systems (add parallelism).
- **Editor:** Create a web-native editor, run the engine inside the browser.

Pill Engine: Roadmap
- **Stateful hot reload:** 🔧 serialize world/resources → reload game DLL → restore state.
- **Archetype-based ECS:** 🔧 extension of the ECS to support archetype-based iteration.
- **Scheduling:** allow for specifying dependencies between systems (add parallelism).
- **Editor:** Create a web-native editor, run the engine inside the browser.

The future looks bright as we have a lot of changes underway. Engine and game hot-reloading with state preservation aim make developers' experience way better. The next big thing is changing the ECS to use archetypes which is a big speedup over current way of doing things. We also plan to allow for having personalized graphs of system dependencies. We want for the engine to be runnable in browser and have a fast and intuitive editor. Lastly, we will strive to optimize the renderer and engine to support mobiles.

- **Stateful hot reload:** 🔧 serialize world/resources → reload game DLL → restore state.
- **Archetype-based ECS:** 🔧 extension of the ECS to support archetype-based iteration.
- **Scheduling:** allow for specifying dependencies between systems (add parallelism).
- **Editor:** Create a web-native editor, run the engine inside the browser.
- **Rendering:** 🔧 Optimized for mobile use-cases.

Pill Engine: Roadmap
- Stateful hot reload: 🔧 serialize world/resources → reload game DLL → restore state.
- Archetype-based ECS: 🔧 extension of the ECS to support archetype-based iteration.
- Scheduling: allow for specifying dependencies between systems (add parallelism).
- Editor: Create a web-native editor, run the engine inside the browser.
- Rendering: 🔧 Optimized for mobile use-cases.

The future looks bright as we have a lot of changes underway. Engine and game hot-reloading with state preservation aim make developers' experience way better. The next big thing is changing the ECS to use archetypes which is a big speedup over current way of doing things. We also plan to allow for having personalized graphs of system dependencies. We want for the engine to be runnable in browser and have a fast and intuitive editor. Lastly, we will strive to optimize the renderer and engine to support mobiles.

Should you write your own engine?

- **You are a wizard Harry:** An engine is: memory management, scheduling, rendering, VFX, Editor. Learn to be a polyglot!

Fun but not for everyone!
- **You are a wizard Harry:** An engine is: memory management, scheduling, rendering, VFX, Editor. Learn to be a polyglot!

An engine spans from the core low-level machinery to VFX and beasts like the visual editor! Be prepared to wear different wizard hats and learn heaps.

- **You are a wizard Harry:** An engine is: memory management, scheduling, rendering, VFX, Editor. Learn to be a polyglot!
- **Iteration time:** You can treat the engine as a playground. Smaller projects → more freedom.

Fun but not for everyone!
- **You are a wizard Harry:** An engine is: memory management, scheduling, rendering, VFX, Editor. Learn to be a polyglot!
- **Iteration time:** You can treat the engine as a playground. Smaller projects → more freedom.

An engine spans from the core low-level machinery to VFX and beasts like the visual editor! Be prepared to wear different wizard hats and learn heaps.

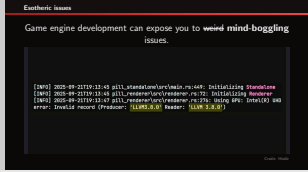
- **You are a wizard Harry:** An engine is: memory management, scheduling, rendering, VFX, Editor. Learn to be a polyglot!
- **Iteration time:** You can treat the engine as a playground. Smaller projects → more freedom.
- **BYO:** Need a new rendering pass? *Sure, just write it yourself and make sure you don't blow up the engine doing that!* ¡4-¿ Ever wanted to test that new way of controlling your character? *Just write it and plug in :)*

Fun but not for everyone!
- **You are a wizard Harry:** An engine is: memory management, scheduling, rendering, VFX, Editor. Learn to be a polyglot!
- **Iteration time:** You can treat the engine as a playground. Smaller projects → more freedom.
- **BYO:** Need a new rendering pass? *Sure, just write it yourself and make sure you don't blow up the engine doing that!* ¡4-¿ Ever wanted to test that new way of controlling your character? *Just write it and plug in :)*

An engine spans from the core low-level machinery to VFX and beasts like the visual editor! Be prepared to wear different wizard hats and learn heaps.

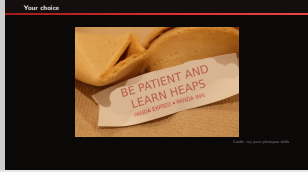
Game engine development can expose you to ~~weird~~ **mind-boggling** issues.

```
[INFO] 2025-09-21T19:13:45 pill_standalone\src\main.rs:449: Initializing Standalone
[INFO] 2025-09-21T19:13:45 pill_renderer\src\renderer.rs:72: Initializing Renderer
[INFO] 2025-09-21T19:13:47 pill_renderer\src\renderer.rs:276: Using GPU: Intel(R) UHD
error: Invalid record (Producer: 'LLVM3.8.0' Reader: 'LLVM 3.8.0')
```

Your choice

Credit: my poor photopea skills

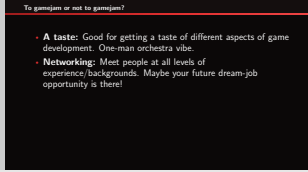


- **A taste:** Good for getting a taste of different aspects of game development. One-man orchestra vibe.

To gamejam or not to gamejam?
- **A taste:** Good for getting a taste of different aspects of game development. One-man orchestra vibe.

To gamejam or not to gamejam?

- **A taste:** Good for getting a taste of different aspects of game development. One-man orchestra vibe.
- **Networking:** Meet people at all levels of experience/backgrounds. Maybe your future dream-job opportunity is there!



To gamejam or not to gamejam?

- **A taste:** Good for getting a taste of different aspects of game development. One-man orchestra vibe.
- **Networking:** Meet people at all levels of experience/backgrounds. Maybe your future dream-job opportunity is there!
- **Experience:** You simply get better at quick iterations, prototyping, and validating your ideas.

To gamejam or not to gamejam?
- **A taste:** Good for getting a taste of different aspects of game development. One-man orchestra vibe.
- **Networking:** Meet people at all levels of experience/backgrounds. Maybe your future dream-job opportunity is there!
- **Experience:** You simply get better at quick iterations, prototyping, and validating your ideas.

To gamejam or not to gamejam?

- **A taste:** Good for getting a taste of different aspects of game development. One-man orchestra vibe.
- **Networking:** Meet people at all levels of experience/backgrounds. Maybe your future dream-job opportunity is there!
- **Experience:** You simply get better at quick iterations, prototyping, and validating your ideas.
- **Motivation:** How good does it feel to have someone play *your* game.

But...

Use Unity/Godot for your first game jams. Only then, start using your own engine/less common ones.

But...
Use Unity/Godot for your first game jams. Only then, start using your own engine/less common ones.

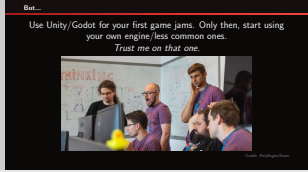
But...

Use Unity/Godot for your first game jams. Only then, start using your own engine/less common ones.
Trust me on that one.

But...
Use Unity/Godot for your first game jams. Only then, start using your own engine/less common ones.
Trust me on that one.

Use Unity/Godot for your first game jams. Only then, start using your own engine/less common ones.
Trust me on that one.

Credit: PolyEngineTeam



Tips

- **Aim small:** Don't write MMORPG, AAA shooter. Instead focus on indie-like vibe. One good gameplay mechanic that is polished is better than a few broken ones.

Tips
- **Aim small:** Don't write MMORPG, AAA shooter. Instead focus on indie-like vibe. One good gameplay mechanic that is polished is better than a few broken ones.

Tips

- **Aim small:** Don't write MMORPG, AAA shooter. Instead focus on indie-like vibe. One good gameplay mechanic that is polished is better than a few broken ones.
- **Use what you have:** Have nice UI components, plug them in. VFX? Camera movement? Bring it on!

Tips
- **Aim small:** Don't write MMORPG, AAA shooter. Instead focus on indie-like vibe. One good gameplay mechanic that is polished is better than a few broken ones.
- **Use what you have:** Have nice UI components, plug them in. VFX? Camera movement? Bring it on!

Tips

- **Aim small:** Don't write MMORPG, AAA shooter. Instead focus on indie-like vibe. One good gameplay mechanic that is polished is better than a few broken ones.
- **Use what you have:** Have nice UI components, plug them in. VFX? Camera movement? Bring it on!
- **One, max two new things during a gamejam:** Stuff does not work, especially when you are tired.

Tips
- **Aim small:** Don't write MMORPG, AAA shooter. Instead focus on indie-like vibe. One good gameplay mechanic that is polished is better than a few broken ones.
- **Use what you have:** Have nice UI components, plug them in. VFX? Camera movement? Bring it on!
- **One, max two new things during a gamejam:** Stuff does not work, especially when you are tired.

Tips

- **Aim small:** Don't write MMORPG, AAA shooter. Instead focus on indie-like vibe. One good gameplay mechanic that is polished is better than a few broken ones.
- **Use what you have:** Have nice UI components, plug them in. VFX? Camera movement? Bring it on!
- **One, max two new things during a gamejam:** Stuff does not work, especially when you are tired.
- **Polish!:** Nothing sells as good as looks! Polished games are the ones we remember.

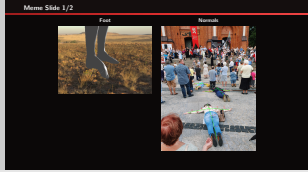
Tips
- **Aim small:** Don't write MMORPG, AAA shooter. Instead focus on indie-like vibe. One good gameplay mechanic that is polished is better than a few broken ones.
- **Use what you have:** Have nice UI components, plug them in. VFX? Camera movement? Bring it on!
- **One, max two new things during a gamejam:** Stuff does not work, especially when you are tired.
- **Polish!:** Nothing sells as good as looks! Polished games are the ones we remember.

Join us!

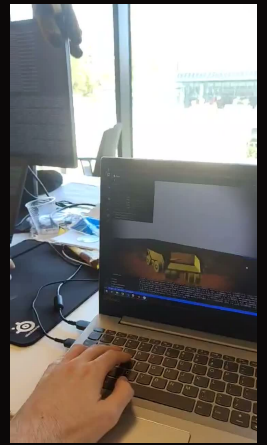
Meme Slide 1/2

Foot

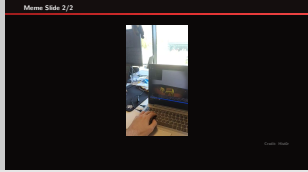
Normals



Meme Slide 2/2



Credit: Hist0r



References

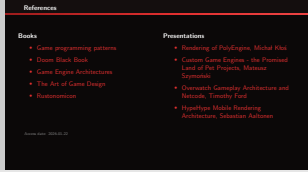
Books

- Game programming patterns
- Doom Black Book
- Game Engine Architectures
- The Art of Game Design
- Rustonomicon

Presentations

- Rendering of PolyEngine, Michał Kłoś
- Custom Game Engines - the Promised Land of Pet Projects, Mateusz Szymoński
- Overwatch Gameplay Architecture and Netcode, Timothy Ford
- HypeHype Mobile Rendering Architecture, Sebastian Aaltonen

Access date: 2026-01-22

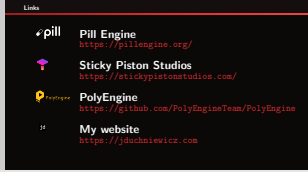


Pill Engine
https://pillengine.org/

Sticky Piston Studios
https://stickypistonstudios.com/

PolyEngine
https://github.com/PolyEngineTeam/PolyEngine

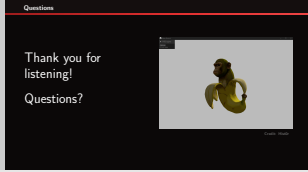
My website
https://jduchniewicz.com



Thank you for listening!

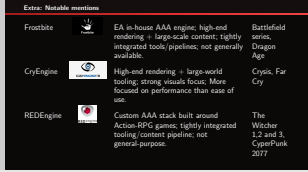
Questions?

Credit: Hist0r



Extra: Notable mentions

Frostbite	EA in-house AAA engine; high-end rendering + large-scale content; tightly integrated tools/pipelines; not generally available.	Battlefield series, Dragon Age
CryEngine	High-end rendering + large-world tooling; strong visuals focus; More focused on performance than ease of use.	Crysis, Far Cry
REDEngine	Custom AAA stack built around Action-RPG games; tightly integrated tooling/content pipeline; not general-purpose.	The Witcher 1,2 and 3, CyperPunk 2077



Some code examples here?

- The biggest problem - ownership and the compiler being very *opinionated*
- Hard to do things *the usual way*, once you are nearly done with a feature, rustc