Particle Workshop V1.0.

Let’s make this show magical.

*Through particles. I’m not a magician. I can’t teach you that.*

*Index*

The possibilities of our Particles -

Shortly explains what the system could be capable of and what this document will cover.

Some theory about Variation and Encapsulation -

Explains the theory behind the solution of the problem we will be facing.

Challenges and Cookies -

Some challenges to apply the theory you read in the other chapter. Everyone who completes the most difficult challenge will receive a gift. No spoilers on what it could be.

*The possibilities of our Particles*

Right now, you will already have a basic particle system[[1]](#footnote-1). It contains an emitter and particles. That’s already great and it can already be very satisfying to play around with it. However, you’ll quickly find its restraints on customizability, such as:

* Being able to assign a spawn region for the particles to spawn in, instead of just from the position of the emitter. See the Appendix for example’s.
* Being able to change the spawn direction of the particles by, for example, limiting it into a certain direction for example.
* Being able to give behaviour to our particle system, such as interacting with ‘the wind’, or point (dis)attraction. Technically: any real particle behaviour.

And some more which are really advanced:

* Being able to accelerate the particles by using the GPU (Graphics Processing Unit, Graphics Card) allowing up to millions of particles per second.
* Adding in complex behaviour systems to, for example, simulate smoke with millions of particles.

This document will aim at solving the first problem: being able to assign customised spawning regions for the particles. In the next chapter, we’ll first explain some theory and techniques surrounding the problem.

The second problem can be solved by yourself using the very same technique that we’ll use for the first problem, but with a different implementation. We cannot guarantee however that we can answer all your questions about this subject, since technically, the workshop does not cover it.

The third point is quite advanced and requires a lot of refactoring (changes) into the code that we’ve made so far. Because this is rather advanced and the aim of the workshop is to educate you all, I decided that it was too much to cover it immediately. It may be covered within another workshop. Keep tab on posters within the Sticky room.

The last two, advanced, points will not be covered by me. I personally can’t do them either. These subjects are simply put (way) too advanced. However, it doesn’t mean it cannot be interesting to know that the possibility exists. If you’re interested, here are some links about these two subjects:

* GPU accelerated particles:
  + <https://www.youtube.com/watch?v=k2wfYzTlHpc>
  + <http://www.catalinzima.com/xna/tutorials/4-uses-of-vtf/particle-systems/>
* Particles living according to models:
  + <https://www.youtube.com/watch?v=DER5xp29z6w>
  + <https://youtu.be/XISqvBVyASo?t=6m22s>

*Some theory about Variation and Encapsulation*

*We’ll discuss the theory required to solve the problem we assigned to ourselves within this chapter. The theory may be hard to grasp at the beginning, however, keep faith, keep reading and keep trying to understand. During this chapter examples will be given to ensure that you will understand. Of course, at any time – feel free to ask questions.*

First off, let’s define the problem that we have and compare it with another problem.

* We want to be able to give the emitter the ability to ‘switch’ between different spawn regions, during runtime.
* We want to be able to change the amount of particles to spawn, during runtime.

The second issue can be solved by assigning a new float value to the variable which keeps track of how many particles should be spawned per second. The code will automatically work with this new value creating the behaviour that we intended (spawning more or less particles).

In the first case however, we’re talking about code changes. Not just a value change. We can’t just ‘change’ code inside our emitter, this isn’t how C# works[[2]](#footnote-2). However, we can change a reference (reference = variable) to the object which contains the functionality that we want. This most certainly sounds quite vague – as I’ve said before: hang on for now.

Both examples given above are forms of **variation**. Variation literally means that what varies. Another example of variation, when looking at our particle system, is the texture the particles use to draw themselves.

We’ve already **encapsulated** some of these variations. Encapsulation means to isolate the variation, to ensure that any changes to the variation will not affect the rest of the code.

For example, the number of particles to spawn per second has been ‘encapsulated’ by making it a variable instead of a hard-coded value. This way, changing this variable will change the desired behaviour (more or less particles to spawn per second), without affecting the rest of the code.

Another example is the texture for our particles. If we want to change the texture of the particles, we assign a new value to the Texture2D variable within the emitter. The emitter will then spawn all newly generated particles with the new texture.

The second example is a crucial one – when you assign a new texture to a variable, you don’t just assign new texture data. You assign a new class and a class contains methods. This principle is also what we want in our original problem – we want to assign new ‘spawn region’ functionality to our emitter during runtime, just as we would assign a new texture during runtime. Keep this principle in mind while covering the rest of the theory.

Before we continue on, it’s important that you properly understand the theory you’ve had so far during the Game Programming or Imperative Programming course. I’m referring to inheritance, overriding and polymorphism. Before you continue, open up the KnowledgeSoFar project and let’s take a look at what these concepts mean.

**Inheritance** is the process of one class gaining all of the member variables and functionality (read: methods) of another class, in addition to the classes own member variables and functionality. This can be seen with the Elite, Grunt, Hunter and Jackal classes. All these classes inherit from EnemyAbstract, allowing them to access data from that class. This can be seen in their constructors. Try it out, by writing “this.” within one of the constructor. You can see all the public and protected variables / methods of the EnemyAbstract class.

**Override** means that you replace the implementation of a functionality (read: a method) from the base class within the sub-class. This can again be seen in all four sub-classes. They all override methods from the base-class. Take note that you can override any method, for as long as the method is either virtual or abstract. The method can have any return type, and any parameter. In this case, the methods are abstract.

**Abstract** means that all sub-classes have to implement the method. This means that every sub-class will most likely have their own, unique implementation.

Before you continue on to polymorphism – ensure yourself that you fully understand what inheritance and overriding means. If you have any questions about these concepts, feel free to ask.

At last there is polymorphism. **Polymorphism** means that you can have a variable that has the type of a base class, but it’s assigned to one of its sub-classes. That sounds quite vague, so let’s check this out within the project: open up the Program class. In here, there is a static type of the EnemyAbstract class. When you open up this class, you can clearly see that it has no implementation. Going back to the Program class, we see that we assign it to a new variable of type Elite.

The Elite class is a sub-class of EnemyAbstract. If we’d play the ‘game’ right now, we’d get the responses that are written within the Elite class. This principle is polymorphism: we’ve assigned a sub-class (the Elite class) to a variable which has the base class (the EnemyAbstract class) as its type. However, when calling out methods or data of this variable, we actually call out the data and / or methods from the sub-class (the Elite class).

Run the ‘game’. Type in some of the commands and see what those commands do within the code. Notice that we’re switching functionality in real time – just like we’d adjust the amount of particles to spawn value (a float value) in real time.

Now, let’s test whether or not you truly understand polymorphism. One way to do this is to write down a very simple program on paper, purely showing the concepts of polymorphism. Another can be to open up Ticktick and find the polymorphism within that project. If you have any questions, or you’d like to know whether or not your assumptions are correct, feel free to ask.

Alright, we’re getting there. Slowly but surely. Let’s line up what problems we’ve seen so far, and some of the concepts we’ve covered so far.

* We want to be able to give the emitter the ability to ‘switch’ between different spawn regions, during runtime.
* We want to be able to change the amount of particles to spawn, during runtime We want to be able to change the texture of the particles, during runtime.

Now, some concepts:

* Inheritance means that one class gaining all of the member variables and functionality (read: methods) of another class, in addition to the classes own member variables and functionality.
* Override means that you can replace the implementation of a functionality (read: a method) from the base class within the sub-class.
* Polymorphism means that you can use a variable that has the type of a base class, but it’s assigned to one of its sub-classes).

Let’s think about this. We want to be able to change the spawning region with the same ease as changing a number or assigning a new texture. We’ve already seen this in the Polymorphism project, where we could assign a new class at runtime. Next off, we want to give every class its own implementation – this can be done by overriding a method. And last but not least, both overriding and polymorphism are dependent on inheritance – all the implementations inherit from the same base-class.

This means we’ll need an base class that has a method with the proper return type and parameter. We want have a position, so the return type needs to be of type Vector2. The should be override able, and thus should be either virtual or abstract so that every sub-class can give its own implementation. Within these implementations, different spawn regions can be defined.

The rest of the implementation I’ll leave up to you. Let’s switch from reading and understanding, to working and implementing. At any time, if you have any questions, feel free to ask.

*Challenges and Cookies*

Alright, now let’s get some work done. Use the theory you’ve read, along with the example project, to create the behaviour that we want: assign new spawn regions with ease. Of course, there are millions of different spawn regions that you can try to implement. However, we can only answer the questions about the following:

- A box spawn region.

- A circle spawn region.

- An outer-box spawn region.

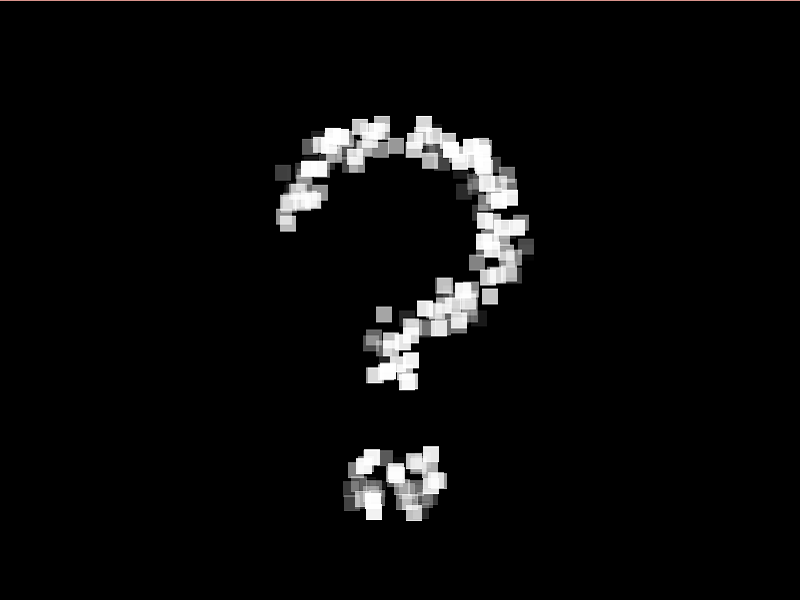
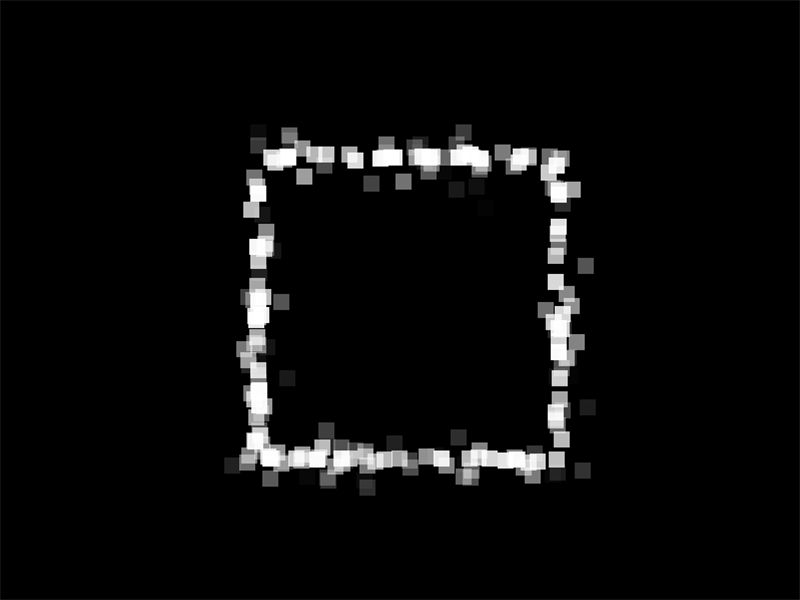
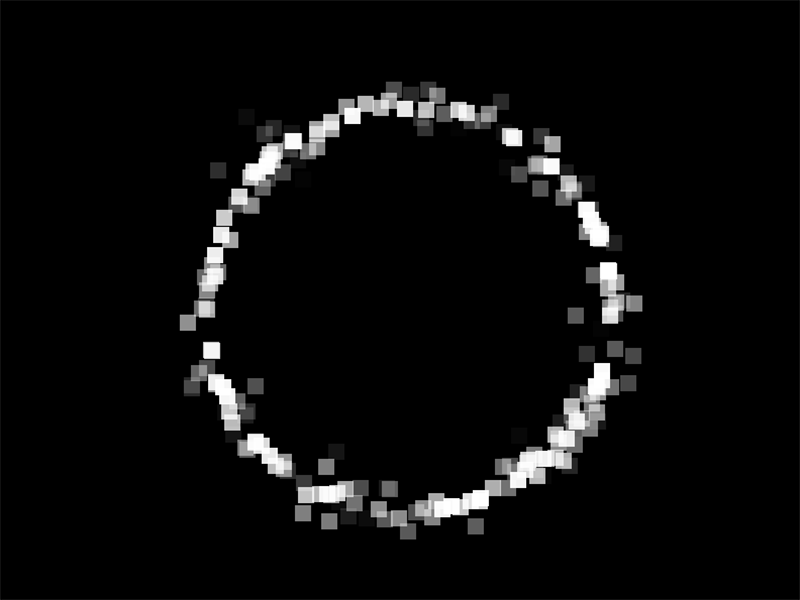
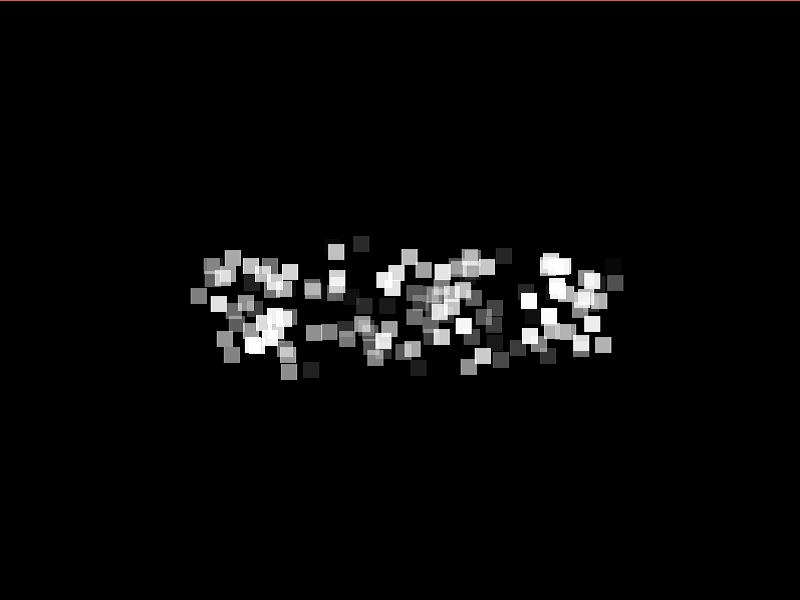
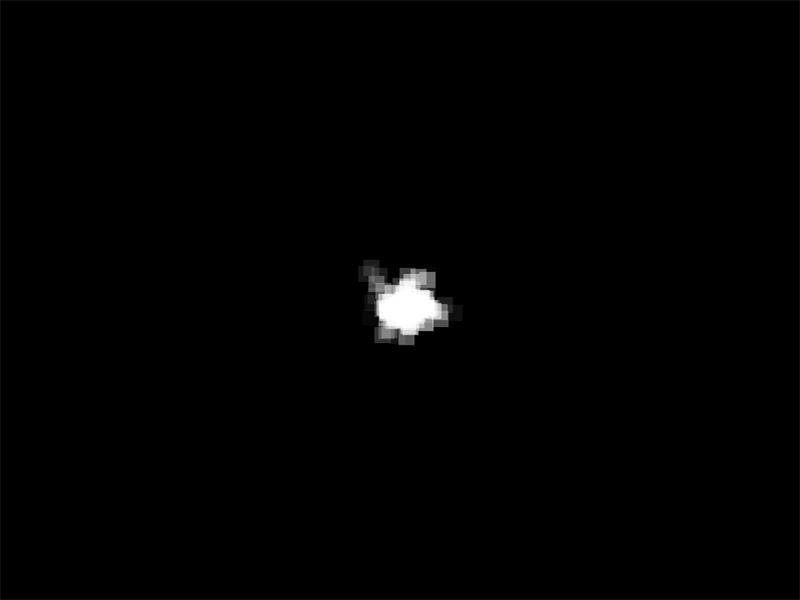
- An outer-circle spawn region.

- A spawn region that is setup in such a way, that it can have any shape.

See the Appendix for images.

Between this chapter and the other is a gap – the actual way to implement this. You have all the theory and the examples to build from to make this implementation, however: sometimes we as human beings just don’t see it. If that’s the case, feel free to ask for further guidance.

Appendix



1. This is a hint. You’re free to read through this document, however, before actually trying out the additions make sure that you got the basic emitter working. [↑](#footnote-ref-1)
2. Technically, this isn’t true. C# does allow you to send through ‘methods’ and thus ‘change’ code. This can be done through delegates. However, we won’t cover these here because they are quite advanced and there is a better way to do it. [↑](#footnote-ref-2)