ESAT INNOVATION SCHOOL | ESCUELA SUPERIOR DE ARTE Y TECNOLOGÍA

Briefing

Nombre Asignatura: Programación Avanzada
Módulo Asociado: 2ºPVG TA
Profesor: Gustavo Aranda
Curso: 2022/2023
Autor/es: Manuel Alcázar López y Pablo Prieto Rodríguez

Índice/Index

1.- Algorithms.

An algorithm is a finite set of instructions carried out in specific order that must be followed for a computer in order to perform a particular task.

There are no well-defined standards for writing algorithms, and they are not written with a specific programming language in mind, but all programming languages share pretty similar basic code structures, such as loops; or flow control such as if-else and so on. An algorithm can be written using these common structures.

Algorithms are usually written step-by-step, but the process of writing an algorithm occurs after the problem has been defined. This is why usually a developer must be aware of the problem for which you are developing the solution. Algorithms are resource-dependent, this means that the programming language and approach used for its implementation will depend on what the developer needs to solve the problem.

When developing our algorithm we have take the following steps:

1. **Obtaining information about the problem:**

Before starting the algorithm we thought it was important to obtain information about the approaches that can solve the problem. This step helps us to get a better understanding of how to get started and how to structure our code in order to provide a useful and efficient solution to the problem.

2. **Analyze the problem:**

Once we have got enough information and decided what programming language we will use for our algorithm, in order to get the best balance between cost and effectiveness, we start analyzing the problem to determine how the algorithm should be structured and the type of data that we will use.

3. **Solving Approach:**

This is, probably, the most important and difficult step when writing an algorithm. The developers have to come up with a problem-solving approach that helps to build the model to solve the given problem. Alongside the development of the code, usually different approaches are to determine which offers the best solution or a better alternative to solve the problem. This makes this step the longest because the basic structure of the algorithm can be constantly changing until the developers come up with the best solution.

4. **Optimize, improve and refine.**

After developing the algorithm we have tried to optimize and improve the code for readability for the team and get a more efficient use of the computer resources.

2.- Programming Paradigms.

- **Procedural Programming:** Procedural programming is based upon the concept of procedure call, a type of routine that contains a series of steps to be carried out.

- **Object Oriented Programming:** OOP is based on the concept of object, which can contain data and code or procedures. The data is usually known as attributes or properties of the object and the code is known as methods. the main feature of the OOP are:

    1. **Encapsulation:** prevent the internal data of the object to be modified from external code.
    2. **Polymorphism:** the methods inside the object can have different parameters.
    3. **Inheritance:** an object can have the methods and attributes from different objects to create a new one.

- **Event Driven Programming**: is a programming paradigm where the events determine the flow of the program. These events are usually related to user input, but they can be different input or outputs from other programs or threads.

The main similarities between these programming paradigms is that they use methods, procedures, routines or subroutines to handle the algorithm's code and behavior and use the basic constructs for data flow and loops that are generally used in programming.

And also the main difference is how they achieve this. Each program paradigm breaks down the program task in different ways. In OOP the task of the program is inside the objects along its data, making the objects its own data structure, while the procedural programming uses procedures, variables and subroutines to operate on data structures. And the event driven programming, generally, uses Finite State Machines, that calls for the different routines that the program must follow for each event.

For our algorithm we have use all of the mentioned programming paradigms:

```
void GameController::input(SDL_Event* e){
  while(SDL_PollEvent(e)){
    ImGui_ImplSDL2_ProcessEvent(e);
    switch(e->type){
      case SDL_QUIT: isRunning_ = 0; break;
      case SDL_KEYDOWN:
        switch(e->key.keysym.sym){
          case SDLK_ESCAPE: isRunning_ = 0; break;
          case SDLK_SPACE: emitter.set_physics(1); break;
          case SDLK_F1: emitter.set_mass(10.0f); break;
          case SDLK_F2: emitter.set_friction(5.0f); break;
        }
      break;
    }
  }
}
```

Here is an example of Event Driven Programming, in which it is easy to identify the different states that the events determine for the program behavior. Unfortunately our application does not need many events, only those who control the main loop state and the window context, so there is little to show.

```
class Path : public Entity {
public:
    Path();
    Path(const Path& copy);
    virtual ~Path();

    void add_vertices(float x, float y);
    void add_vertices(const Vec2& vert);

    void loadSquare();
    void loadCircle();
    void loadStar();

    void set_color(SDL_Color color);
    void lerpUnclampedColor(SDL_Color lerp, float time);
    void draw(const WindowController& wc) override;

    std::vector<Vec2> vertices() const;

private:
    std::vector<Vec2> vertices_;
    SDL_Color color_;

};
```

This is an excellent example of OOP, the objects are instances of data structures called classes. Here we can observe the attributes of the object, in the bottom that remain private and the methods above that are public, this is the so-called encapsulation that prevents the modification of the object data. Next to the class declaration we can see how this object inherits from another class called Entity, which means this class will have the attributes and methods of Entity plus its own. And finally we can see some keywords such as virtual and override that are related to polymorphism and virtual inheritance.

```
void PathWindow(Path* path){

    static Vec2 position = path->position();
    static Vec2 scale = path->scale();
    static float rotation = path->rotation();
    static int color[3] = {0xFF, 0xFF, 0xFF};

    ImGui::Begin("Path Window", nullptr);

    ImGui::Text("Position X"); ImGui::SameLine();
    ImGui::DragFloat("##p1", &position.x);
    ImGui::Text("Position Y"); ImGui::SameLine();
    ImGui::DragFloat("##p1", &position.y);

    ImGui::Text("Scale X"); ImGui::SameLine();
    ImGui::DragFloat("##s1", &scale.x);
    ImGui::Text("Scale Y"); ImGui::SameLine();
    ImGui::DragFloat("##s1", &scale.y);

    ImGui::Text("Rotation"); ImGui::SameLine();
    ImGui::DragFloat("##r1", &rotation);

    ImGui::DragInt("R##r1", &color[0], 1, 0, 255, NULL, ImGuiSliderFlags_AlwaysClamp);
    ImGui::DragInt("G##g1", &color[1], 1, 0, 255, NULL, ImGuiSliderFlags_AlwaysClamp);
    ImGui::DragInt("B##b1", &color[2], 1, 0, 255, NULL, ImGuiSliderFlags_AlwaysClamp);

    path->set_position(position);
    path->set_scale(scale);
    path->set_rotation(rotation);

    // path->set_color((Uint8)color[0], (Uint8)color[1], (Uint8)color[2]);

    ImGui::End();
}
```

For the implementation of the GUI, using ImGui, we have used procedural programming, which uses references to the objects to execute a finite set of instructions. Each class could have its own GUI method, but this would have increased the code, while using the OOP inheritance and procedural programming we can reduce the amount of work and code for the algorithm and improve the application at the same time. In this example we can see the different steps the procedure will follow once the program calls to the function.

3.- Implementation and Debugging.

An Integrated Development Environment (IDE) is a software application that provides comprehensive tools for software development. Usually IDEs consist of a source code editor, a compiler or build automation tool and a debugger. Some of them also include a version control system for the source code, or possess a way to easily implement one.

Working with an IDE improves the software development process by creating a space where the most common tools for developers are fully integrated and increase the efficiency of the development process and the workspeed of the developers. If we combine an IDE with an standar Normative Programming the workflow of the development becomes similar to all developers, which reduces the production time of the product and ensures a reliable code application. It also improves code management and scalability, which greatly helps software development enterprises. Working without an IDE and a normative, difficulties the software development, increases the production time and its cost, and makes the algorithms less reliable and scalable.

Using both, IDE and a normative, has helped us greatly to implement our code. All the team members have used similar syntax when implementing the code and the use of an IDE, Visual Studio 2019 in our case, to edit, compile and especially debug our code in the same workspace makes the development far easier, especially when using the debugger to detect failures and crashes in the code. The debugger has helped us to detect the main problems we had to face when developing the application, instead of trying to figure out where the problem is, if not using a debugger.

The steps for debugging an application using Visual Studio are:

1. **Attach the process to the debugger:**

Debugging in Visual Studio occurs automatically when you run from Visual Studio with F5 or select Debug -> Start Debugging. Alternatively, you can attach to a running process with Debug -> Attach to Process (Ctrl + Alt + P). If you are compiling from the command prompt use the command devenv with the name of the executable and the files to debug.

2. **Debugger Break Mode:**

The debugger has to be in “Break Mode” for debugging. That means the program is currently paused on a specific line of code. The program also needs to have the debugging symbols loaded, this can be done by compiling the code in debug.

You can get to break mode with Debug | Break or by placing breakpoints. We are usually going to use breakpoints because in most debugging scenarios we will want to debug when the program reaches a certain line of code

You can place breakpoints by clicking on the margin, pressing F9, or Debug | Toggle Breakpoint. Once set, when your program reaches that line of code, Visual Studio’s debugger will stop execution and enter “Break Mode”:

When in break mode, the yellow line represents the next line of code to execute and you can debug interactively and see how your execution of code progresses. The basic features of code navigation are:

- **Continue (F5):** will quit break mode and continue the program’s execution until the next breakpoint is hit, entering break-mode again.
- **Step Over (F10):** will execute the current line and break on the next line of code.
- **Step Into (F11)** is used when the next execution line is a method or a property. When clicked, the debugger will step into the method’s code first line. By default, properties are skipped. To enable stepping into property code go to Tools | Options | Debugging and uncheck Step over properties and operators.

The debugger also possesses different windows that will display information about the current state of the code and the memory of the program, with which we can investigate the value of the program’s variables or the call stack of the program.

4.- Personal Tasks and Workgroup Plan.

1. **Personal Tasks:**

Base hierarchical classes and GUI: Pablo Prieto Rodrigez.
- Entity.
- Path.
- Texture.
- Game Manager.
- ImGui Controller.

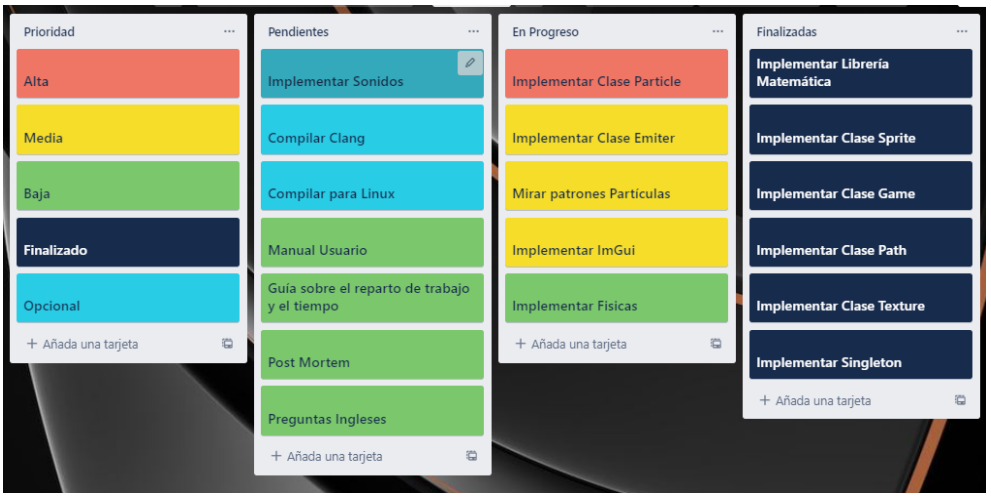
Documentation:
- User Manual.
- Personal Tasks and Workgroup Plan.
- Post Mortem.

Particle System: Manuel Alcazar Lopez.

- Particle.
- Emitter.
- Emitter Pool.
- Collider2D.
- Sprite.
- Window Controller.
- Game Controller.

Documentation:

- Algorithms.
- Programming Paradigms.
- Implementation and Debugging.

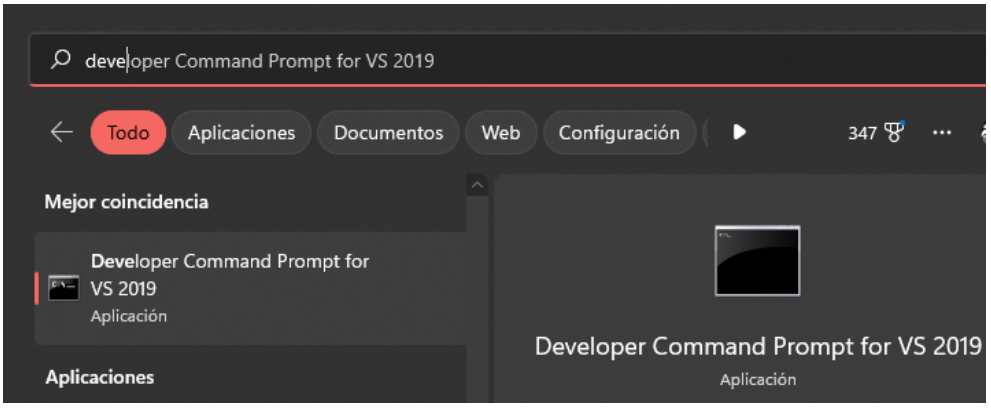


We didn't have an exact time division of the tasks, instead we have used trello to prioritize the tasks that we should have completed before advancing into other tasks or that were more important to implement than the others. We have just simply followed the priority order of the task.

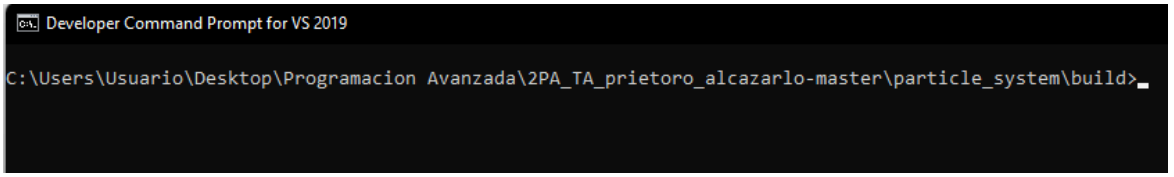
5.- User Manual.

5.1 Execution manual

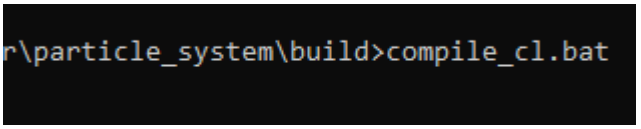
To run the demo, we must open the Developer Command Prompt for VS2019 or similar application to compile and execute the program:



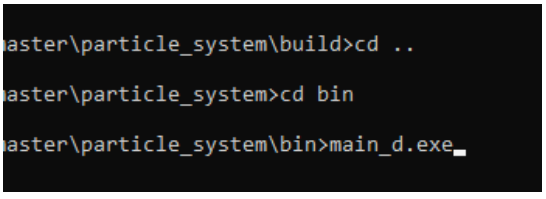
Once opened, access the unzipped folder and access the build folder:

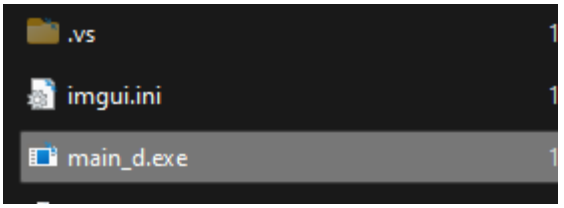


Then, compile the program writing ***compile_cl.bat*** and pressing the enter key:

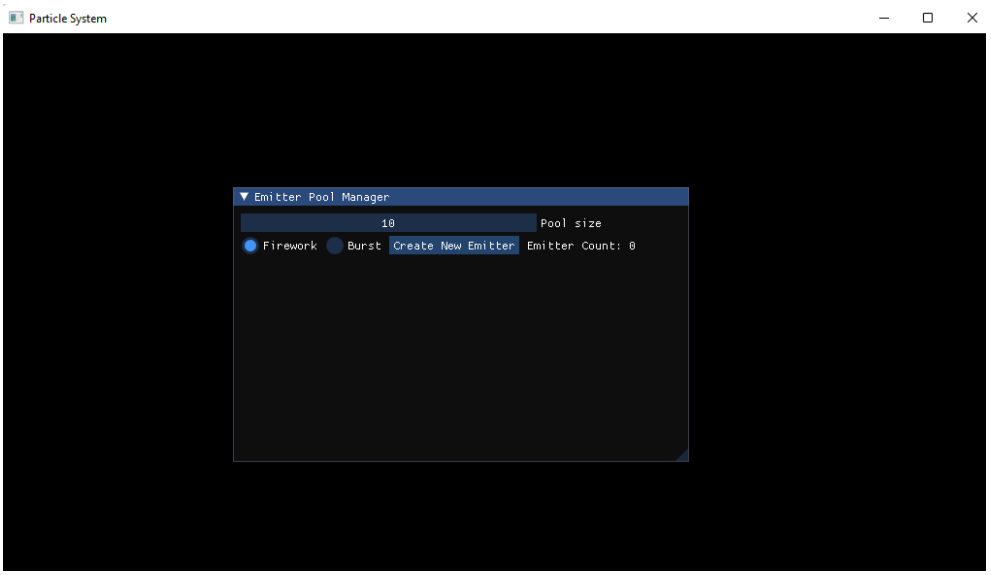


Now you should see some text related to the compilation process without much importance. Once finished, the executable will be available in the **bin** folder of the project named **main_d.exe**.
You may either execute it from the command window or by double clicking on the file directly:





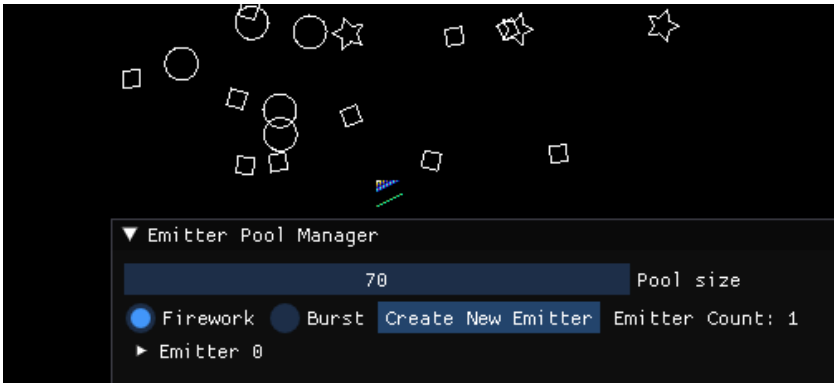
Once executed, the program will start a window called ***Particle System***:

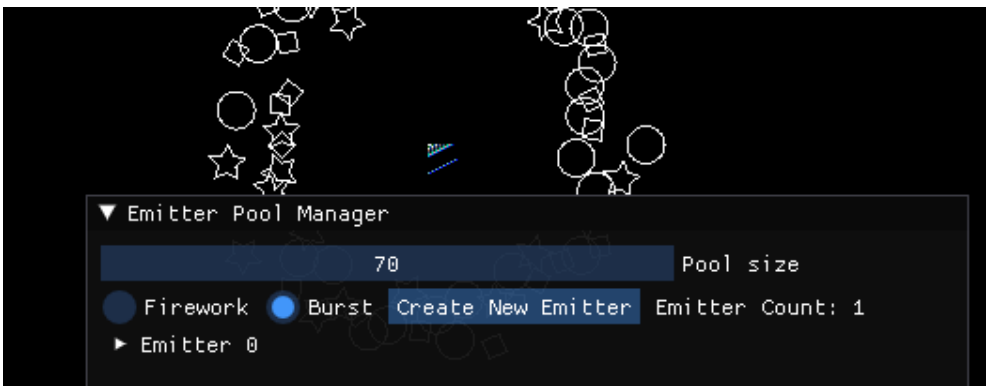


We can now take a look to the ImGui interface, where we find some options, such as:

- **Pool size:** This slider allows us to determine the size of the particle's pool, which means, the total number of particles that the emitter will use. The minimum value is 10 particles, and 128 is the maximum.
- **Firework and Burst:** These buttons select the particle mode.
- **Create new emitter:** This button allows you to create a new emitter and place it in any part of the window by clicking wherever you want. The text next to this button is the emitter's counter.

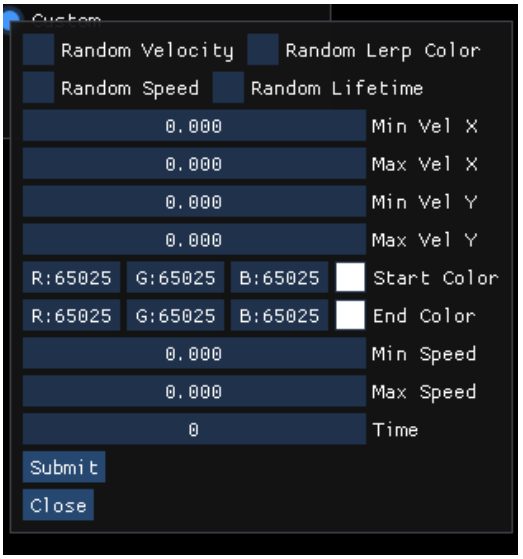
Let's see a couple examples:





Now that our emitter is created and its particles are flying around the window, the emitter options are now available:

From here, we can now change the emitter's size using the **slider** and clicking on the **set size button**. We can also change the emitter's mode and even customizing it:



We are able to customize the velocity in the axis, the speed of the particle, its lifetime and the starter and final color. Once selected the desired values, click on submit button to save the changes.

6.- Post Mortem.

The basic hierarchical implementation of classes has been easy to implement, with no further issues. The math library, although it passed the unitary test, the test may be wrong because some Matrix4 and Matrix3 operations for transformations had to be changed in order to draw and transform 2D and 3D objects properly, and it has still passed the test. So, our first problem was to determine if the matrix has to be in row-major order or in column-major order, just to finally be a mix of both, because of the unitary test.

Our second major issue was to develop a sustainable particle system demo. The first attempt was to develop a particle system with dynamic memory allocation but low speed of the application, plus the issues to merge the particle system with an understandable GUI and difficult to scale the code, this attempt was discarded. In the second attempt, the approach was more easy to implement and scale, due to the code having a fixed pool size of particles that cannot change in execution time.

From here we start to scale the code, by adding more features to the application while focused on making an easy implementation for the GUI. The code had to be refactored once again to have, as said, an easy implementation, by using a single struct to manage all the information the particle has to receive from the emitter, this also allows us to implement a semi dynamic allocation of the different particle pool that the emitter uses. We can say we are satisfied with the final implementation of the demo.