C++ SV Project

Jeremy Mion (261178)

December 11th 2018

Q.1

Q.1.1

The only logical thing to di is to create a private methode. In my case “clamp”

Q.1.2

The reason that the default = and copy constructor are sufficient is that they both do a surface copy of the elements. Codding them by default allows the user of our class to know that they can safely use these constrcutors.

Q.1.3

A double loop will allow for a more concise piece of code. There is no point in describing it in gory details in this file. See the actual code for implementation details.

Q.1.4

Taking a const reference is almost always better. The only case where we decided to not use a const ref, is in the constructor since we want a copy of the vector to operate on it.

Q.1.5

All the methodes that do not affect the Object should be constant. This is extremely important since C++ is strict on how it manages const functions. To see which ones where declared as const see the CircularCollider.hpp file.

Q.1.6

We are simply adding another way to call : we are not writing code that has any functionality. This is important because it allows use to ensure that all of the functions that are designed to have the same behavior behave in the same way. isCircularColliderInside(other) isColliding(other); isPointInside(point);

Q.1.7

I chose an internal definition for most of the functions because there equivalent in non operator function calls are also internal.

Q.1.8

For which functions are receiving parameters by reference see .hpp. In all cases where we do not need a copy of the object to operate on it is better to use a const reference since it will improve performance.

Q.1.9

See CircularCollider.hpp for the list on functions that we decided to declare as const.

Q.2

Q.2.1

The draw method should be const. The other ones will make modifications to the environment, therefor they cannot be constant.

Q.2.2

Use the delete of the operator= and the copy constructor.

```
/*!
 * Disabeling copy constructor since this is a large object that should not be copied.
 */
Environment(const Environment&) = delete;
Environment& operator=(const Environment& env) = delete;
```

Q.2.3

This means that when the Environment is destroyed it needs to destroy all the Animals.

Q.2.4

Warning to check with others.

The definition of the use of the different keys is done in the EnvTest

HELP HELP HELP

Q.2.5

Well we clearly want one method that will calculate the force that is being applied to the automaton. This method called attractionForce will return the force that the automaton is experiencing. This will allow classes inheriting of ChassingAutomaton to simply redefine the attractionForce and have the wanted behavior. The methode prototypes are as follows:

```
/*!
 * Calculates the attraction force that the automaton is experiencing
 * @return force that are being applied to the robot.
 */
Vec2d attractionForce() const ;

/*!
 * Makes the automaton moved based of of the force that it is experiencing.
 * @param force that the robot is experiencing
 * @param dt time that has passed since previous update
 */
void updateMovementVariables(const Vec2d& force, const sf::Time dt );
```

Q.2.6

We will for now declare a default value in the constructor of ChassingAutomaton. We have added it to the consctructor with a default value. We will adapt how this enum is set, once we have a better grasp of the context in which it would need to be changed. The enum names are of course capitalized to respect the convention that constants are written in capital letters. The main issue with using enums is of course that we cannot assign floating point values to the elements of an enum. Therefor we had multiple possibilities that where present. Store the enum as integer values by multiplying by a factor 10. This is far from

ideal since it means that there is information that is encoded into the enum but that can be misinterpreted. The solution that we decided to apply in this case is to use a private methode to resolve the enum value to it's floating point equivalent.

Q.2.7

The animal is a seperate entity. Therefor we can imagine that classes that extend the Animal class will want to influence the rotation of the animal. There is no reason why an external class such as the environment should be able to influence the rotation of the automaton. For the method setPosition does not exist since I have not had any use for it so far. .

Q.2.8

In the method draw of the Environment we need to call the draw of Animals.

Q.2.9

This is done to break encapsulation to be able to test the animal class by setting the rotation. We need this for the test.

Q.2.10

A list of targets. We are using a list because it allows use to return as many elements as we want. At this point in time since we are only intrested in targets a list of Vec2d should be enough even though we can forsee that in the future we might need to change this to pointers towards targets.

Q.2.11

We simply need to update the animals from the update methode of environment. This is how most games and simutations are build where the environment is in charge of updating when it wants all of the objects that are evolving within it.

Q.3

Q.3.1

All of these methods are declared as abstract : *etStandardMaxSpeed, getMass, getRandomWalkRadius, getRandomWalkDistance, getRandomWalkJitter, getViewRange et getViewDistance* . We need to use the override keyword every time that we want to redefine one of the abstract methods. Here since we clearly do not know the implementation of these methods in the Animal class we will force the children of the class to define the behavior that they will provide.

Q.3.2

We need to change the line where energy is indicated. For example:

```
"energy":{
                "initial":80,
                "min mating male":450,
                "min mating female":800,
                "loss mating male":200,
                "loss female per child":100,
                "loss factor":0.001
            },
```

Q.3.3

The classes Updatable and Drawable are being used as interfaces. The concept of interfaces does not exist in C++ but can be mocked with a 100% abstract class. The idea is that all objects that can be drawn can be seen as drawable objects. It is a contract that all subclasses have to fulfill. In our project we will consider that a CircularCollider is Drawable. The idea of implementing at this level is that all the subclasses can and should call the implementation of there parent class. They will therefor all inherit of the possibility of drawing a visual representation of the collider in debug mode. In our project we will assume that we can have circularCollider's that are static object and therefor don't need to be updated. We will consider that all OrganicEnity are Updatable.

Q.3.4

It a way to build a program not makes modifications not easy due to the lack of maintainability. Removing or adding new classes require a large amount of work to rework existing code to add new tests. Using a double dispatch is a much

better way of doing things since it relinquishes the control of who can eat what to the class itself.

Q.3.5

It is interesting to implement the draw method for the circular collider in it's own class. It makes sense that a circular collider is in charge of displaying itself for debugging purposes. We therefor changed the inheritance of Drawable, moving it up from OrganicEnity to CircularCollider. For the moment there is no specific need / or logic to justify the circular collider having an update method. Therefor the implementation of the Updatable "interface" will remain in Animal. A explicit call to CircularCollider::draw needs to be made at the beginning of the draw method of any children to enable the drawing of the circular collider.

Q.3.6

The best implementation in this case is to provide 2 methods that are used in the default implementation of isDead(). These methods will allow access to the longevity and min_energy_level. Of course we will declare them as virtual and allow any of the classes that implement to change the definition of what the default values are. In this way the method to determine if the organicEnity is dead can remain in the OrganicEnity class and the sub classes have the possibility to modify it's behavior by changing the getters.

Q.3.7

When the OrganicEntities die of old age the environment needs to free the memory that was allocated when we created the animal. This is crucial to avoid memory leeks. The environment is in charge of checking at each update cycle if the entity is dead and unregistering them. If this task becomes a large overhead one would imagine using a queue for every type of animal. This would improve performance significantly if there is a lot of animals being simulated. We will consider this in case performance becomes an issue.

Q.3.8

First of all we we need to define the condition when the Animal will be starving. This information will be used in getMaxSpeed. For the definition of when an animal is starving we decided that each animal can be starving at a different energy level. For proper use we therefor added an *scorpion_energy_starving* and *gerbil_energy_starving* to our app JSON config file. We used a virtual method to define the updateState leaving the choice of the value to return up to

the non abstract animals. With this solution we are respecting the app design and choice of keeping all of these parameters modifiable at run time. We also added *animal_starving_speed_factor* to the JSON because it is the only logical place where such a setting should go. We do not want to affect the state that the animal is in since we would like the animal to behave is it normally does but in a weaker form (moving less fast).

Q.3.9

We could have decided that the circular collider provide the code to deal the all the collisions. The issue with this type of implementation is that all of the code is in one class that becomes quite large and dependant on other classes. The architecture chosen allows for a more flexible implementation.

Q.3.10

Exactly like for mate we use double dispatch to have the type of both this and mate. We define method *meet* and *meetManagement*. *Meet* calls *meetManagemetn* which takes care of dealing with the animals matting. The matting involves changes to the animals matting. Since all the fields are defined in animal we have defined a *procreate* method that will take car of making the appropriate changes to the animal.

Q.3.11

We will put in place the gestation time by creating a sf::Time variable that will be used to track the amount of time that the animal has been in gestation.

Q.3.12

The method give birth is defined in the *Animal* class and all of it's children. The definition in the *Animal* class only determines if the animal can give birth. It is another check to ensure that no male animals give birth. This check should be redundant since this method should only be called by females. But to avoid missus of this method in the future we felt like the proper way of defining this giveBirth method was to provide a check.

Q.3.13

The number of babies is stored in the animal. This attribute has a getter provided for it ensuring that if a male animal calls the method it will return 0. It is logical that a male cannot have children.

Q.3.14

It is better to avoid storing pointers to organic entity in the animal if possible. This would provide storage for pointers that could be used when the organic entity disappears from the environment. We instead decide to store a list of locations where we have last seen predators. There are some cons to this implementation since if a animal sees the predator twice in different locations it will register it as another location where there is prey without removing to location where it previously saw the preadator. We can argue that this implementation makes sense since a Gerbil would probably not be able to tell apart scorpions.

Q.4

Q.4.1

A wave should inherit from a circular collider and it would be updatable, drawable. We do not implement the drawable interface since a circularCollider already does.

Q.4.2

A wave does not need to keep track of time. Since it would be easier and logical to offer the possibility to a circularCollider to change it's radius this functionality was added in the circularCollider class. We therefor only need to update the radius when a call to update is made.

Q.4.3

An environment will have a list of waves* that will be updated, drawn, and deleted. We also need to add a method to add a wave. And of course not forget to clear the remaining waves from the list at the destruction of the environment.

Q.4.4

We need to create a method addObstacle. Modify the *Environment* delete, draw, methods to take the obstacles into account.

Q.4.5

The neuronalScorpion will have an array of pointers to sensors. The sensors will store there relative angle to the scorpion. This is not the best idea when we are

looking at the architecture of the NeuroanlScorpion and it's sensors. Making the assumption that a sensor needs to be attached to an other entity is OK and we will make this compromise. This means that a sensor knows is relative angle from the scorpion.

Q.4.6

```
Vec2d NeuronalScorpion::getPositionOfSensor(const Sensor* s) const;
```

A neuronal scorpion need only to know what sensor it is calculating the position for. Since the sensor stores it's own angle we do not need to look up, for example in a map where the sensor is positioned. Instead we can simply access the sensors angle by calling a *getAngle()* method from the scorpion.

Q.4.7

We will add the environment the following method:

```
std::list<Wave *> getWaveCollidingWithSensor(const Vec2d& location ) const;
```

We will create a method that will only return the waves that are currently colliding with the sensor. A critic of our current implementation is that we are dealing a bit to much with the calculation of which waves are actually colliding. The advantage of this is that we are not sharing useless pointers. We are willing to delegate more of the work that should be up to the sensor to decide to the environment since it allows use to preserve good encapsulation of pointers.

Q.4.8

Both classes are quite tightly coded. Meaning that sadly we might have issues making another animal use sensors. The best solution here would be to use an intermediate class that enforces that the neuronal animal has all of the methods that the sensor needs. Both classes are tightly integrated since sensors needs to know it's owners, and some of it's neighboring sensors. There is a lot of sharing of pointers which isn't optimal for code independence but will improve performance.

Q.4.9

Since a sensor has a list of sensors that it needs to inhibit, we need to use a method to initialize all of the sensors and link them together. For this purpose we wrote the method : *void initializeSensors()*.

Q.4.10

We will use an enum to model the different states. Again we made the choice to use our magic macro to do so because it takes care of the toString method as well.

Q.4.11

We will have a stateTimer in the NeuronalScorpion that will be tasked with doing the time keeping.

Q.4.12

Each time we change states we will reset the **stateTimer**. The **stateTimer** will be incremented every time update is called and that we are in *MOVING* or *IDLE*.

Q5

Q.5.1

It is the purpose of a map to associate key to values. Since the application is using key to decide which graph to show it is natural to use a map to store the graphs that are tied to the keys. A map only allows one instance of a object with the same key. A vector is simply not appropriate for such a use. We could of course use a vector but this would involve coding a wrapper around it to make it behave like a map. Lets use what is already done for use and use the one that the standard library has :)

Q.5.2

We have made the choice to add quite a bit of mechanics to all the classes that we want to monitor. We did this because it allows for almost no overhead when we are updating the graph. The environment has a mad that counts the number of

Q.5.3

In our design we do not need to do any special treatment to have it behave properly. Since the stats are tied to an environment switching environment will switch stats as well.

CONCEPTION

I added a macros utility file where I will put all the macros that will be utilities for this project. This includes for now the parameters to suppress unused attributes for GCC. This is done here to allow flexibility in case we change compilers since the suppress warning flags are not part of the C norm but are specific to GCC.

I decided to structure the obstacles such Rocks into a abstract class that defines solideObstacles and let Rocks be an instance of a solideObstacle. This allows for easy definition of other types of solid obstacles. Some changes where made to the WaveTest.h . To be specific we changed the typedef of Obstacle to SolidObstacle instead of leaving as defined in the handout as a CircularCollider. This means that in our definition only solidObstactles will have the effect of breaking a wave.