How do I?

A guide about creating a **core**, a **game** and a **display**

Written by Virgile BERRIER

CORE

The core is what will link both your Game and your Display. More than that, it's what runs these two. Regardless of what is your method of managing Shared Libraries, you should store your **Game instance** as a "GameModule" (see IGameyModule.hpp) and your **Graphical instance** as a "DisplayModule" (see AdisplayModule.hpp).

Both are unique pointers of the IDisplayMdule and IGameModule **virtual classes**. Then you just use the entryPoint function from the loaded libraries to generate an instance.

If you need to know the type of a library, you can still use the getInformations function. (see Shared/LibraryType.hpp) It should return a structure containing 2 members :
- type, containing shared::library::librarytype::type::GAME or shared::library::librarytype::type::GRAPHICAL.
- isInvisible, containing the visibility of the library. If isInvisible is true, It means that it should be ignored by the menus when listing games and libraries.

Once your instances are loaded, you can declare **3 variables** :
- a shared::**Display** (see Shared/Display.hpp)
- a shared::**Inputs** (see Shared/Inputs.hpp)
- a shared::**Meta** (see Shared/Meta.hpp)

In addition to that, you should also set a **default** value for your window, in tile :

```
| display.screenSize( ).x = 20; display.screenSize( ).y = 20;
```

You can now start your main loop, in the following order :
- START OF THE LOOP
- updateInputs function from your DisplayModule
- updateFrame function from your GameModule
- display function from your DisplayModule
- END OF THE LOOP

In addition to these steps, you'll have to add 3 additional elements to your loop :
- a way to put your **refresh rate**, in **microseconds**, inside your shared::Inputs :

```
        | inputs.delta( ) = [ . . . ];
```

- a way to manage the data contained in your shared::Meta :

```
        | std::string loadGame = meta.extractGame( );
        | if (!loadGame.empty( ))
        |         [ . . . ]
        | std::string loadLibary = meta.extractLibrary( );
        | if (!loadLibary.empty( ))
        |         [ . . . ]
```

- a way to manage additional inputs that are used by the core :

```
        | if (inputs.hasBeenPressed(shared::inputs::EXIT)) [ . . . ]
        | if (inputs.hasBeenPressed(shared::inputs::NEXT_LIBRARY)) [ . . . ]
        | if (inputs.hasBeenPressed(shared::inputs::NEXT_GAME)) [ . . . ]
```

| if (inputs.hasBeenPressed(shared::inputs::RESTART)) [. . .]
| if (inputs.hasBeenPressed(shared::inputs::MENU)) [. . .]

LIBRARIES

Here's an example for an empty "test" game module :

include/**Test**.hpp :

```
| #include "AGameModule.hpp"
| class Test : public AGameModule {
| [...]
| }
```

src/Test/**entryPoint**.cpp :

```
| #include "Test.hpp"
| extern "C" GameModule entryPoint(void) { return createGameModule<Test>( ); }
```

src/Test/**Constructor**.cpp :

```
| #include "Test.hpp"
| Test::Test(void) { name = "Test"; }
```

+ the provided src/LibraryType/**Game**.cpp

This code shouldn't do anything useful, but it can still be loaded.

Let's break up what's happening : First and foremost, the Core can load information about "test" with the getInformations function contained by src/LibraryType/**Game**.cpp. In addition, the constructor sets the name of the object for the getName function.

The include/**Test**.hpp is used to store the class of the game. In addition of overriding AGameModule, it can use its own members.

Finally, the entryPoint function uses the createGameModule<> function from IGameModule.hpp (itself from AGameModule.hpp) to allow the Core to generate an instance of the game.

GAME LIBRARIES

Game libraries are simple, in theory. They only use one function : updateFrame.

You must provide both a shared::Display and a const shared::Input. Alternately, you can also add a shared::Meta if you need it.

Here's an example for our "test" game module, **in addition to the previous code** :

include/**Test**.hpp :

```
| #include "AGameModule.hpp"
| class Test : public AGameModule {
| private:
|         shared::DisplayedObject _cursor;
| public:
|         void updateFrame(shared::Display &display, shared::Inputs const &inputs) override;
| }
```

src/Test/**Constructor**.cpp :

```
| #include "Test.hpp"
| Test::Test(void)
| {
|         name = "Test";
|         _cursor.shape( ) =  shared::shapes::SQUARE;
|         _cursor.mainColor( ) =  shared::color::WHITE;
| }
```

src/Test/**updateFrame**.cpp :

```
| #include "Test.hpp"
| void Test::updateFrame(shared::Display &display, shared::Inputs const &inputs)
| {
|         if (inputs.hasBeenPressed(shared::inputs::UP)) _cursor.position().y -= 1;
|         if (inputs.hasBeenPressed(shared::inputs::DOWN)) _cursor.position().y += 1;
|         if (inputs.hasBeenPressed(shared::inputs::RIGHT)) _cursor.position().x -= 1;
|         if (inputs.hasBeenPressed(shared::inputs::LEFT)) _cursor.position().x += 1;
|         display.list( ).clear( );
|         display.list( ).push_back(_cursor);
| }
```

First in first, this code should be useful to test things, displaying a controllable cursor on screen.

Here, we're storing the cursor directly as a "DisplayedObject", itself as a member of the "Test" class. Generating an instance of Test will generate a paired _cursor. Whenever the constructor is called, we initialize additional data for DisplayedObject : here, we define no sprite but instead a WHITE SQUARE shape.

Finally, our updateFrame will detect whenever we press a correct button, updating our cursor. After that we just clear then refill the DisplayedObject list from the &display object.

To help you, here's a table explaining all the ways you have to check your inputs.

Function	*Inactive*	*Just pressed*	*Held*
inputs.**isPressed**( )	FALSE	TRUE	TRUE
inputs.**isntPressed**( )	TRUE	FALSE	FALSE

inputs.**hasBeenPressed**( )	FALSE	TRUE	FALSE

Going further, we'll expand our updateFrame :

src/**updateFrame**.cpp :

```
| #include "Test.hpp"
| void Test::updateFrame(shared::Display &display, shared::Inputs const &inputs)
| {
|         shared::DisplayObjectSpritePosition previousPosition =
|                 {_cursor.position().x, _cursor.position().y};
|         if (inputs.hasBeenPressed(shared::inputs::UP)) _cursor.position().y -= 1;
|         if (inputs.hasBeenPressed(shared::inputs::DOWN)) _cursor.position().y += 1;
|         if (inputs.hasBeenPressed(shared::inputs::RIGHT)) _cursor.position().x -= 1;
|         if (inputs.hasBeenPressed(shared::inputs::LEFT)) _cursor.position().x += 1;
|         if (previousPosition.x != _cursor.position().x &&
|                 previousPosition.y != _cursor.position().y) {
|                 display.fingerprint( ) += 1
|         } else {
|                 display.list( ).clear( );
|                 display.list( ).push_back(_cursor);
|         }
| }
```

Fingerprint? So, what is that? It's a value use by GameModules to indicate to DisplayModules that display hasn't been refreshed. Therefore, in this case, if the the position hasn't been changed, display.**fingerprint**( ) gets incremented by +1, so our Graphical Library doesn't need to refresh.

GRAPHICAL LIBRARIES

Sorry but I still haven't written this part.

All I can give you for the moment is this table listing every inputs :

Name	*Inputs*	*Actions*
shared::inputs::**UP**	Arrow up	*Game specific*
shared::inputs::**DOWN**	Arrow down	*Game specific*
shared::inputs::**LEFT**	Arrow left	*Game specific*
shared::inputs::**RIGHT**	Arrow right	*Game specific*
shared::inputs::**A**	Space bar	*Game specific*
shared::inputs::**B**	X	*Game specific*
shared::inputs::**START**	W	*Game specific*
shared::inputs::**SELECT**	C	*Game specific*
shared::inputs::**MENU**	M	Go back to the main menu

shared::inputs::**RESTART**	R or Shift+1	Restart the current game
shared::inputs::**NEXT_GAME**	G or Shift+2	Swap the Game
shared::inputs::**NEXT_LIBRARY**	L or Shift+3	Swap the Graphical Library
shared::inputs::**EXIT**	E, Q or Shift+4	Exit Arcade

If you want to use them, you'll need to have something like this :

src/TestGraphical/**updateInputs**.cpp :

```
| #include "TestGraphical.hpp"
| void TestGraphical ::updateInputs(shared::Inputs &inputs)
| {
|         inputs.open( );
|         if ( [ . . . ] ) inputs << shared::inputs::UP;
|         if ( [ . . . ] ) inputs << shared::inputs::DOWN;
|         if ( [ . . . ] ) inputs << shared::inputs::LEFT;
|         if ( [ . . . ] ) inputs << shared::inputs::RIGHT;
|         [ . . . ]
|         if ( [ . . . ] ) inputs << shared::inputs::EXIT;
|         inputs.close( );
| }
```