How to create a graphical library

Arcade documentation

April Revision

1 — First steps

Your graphical library will take the form of a shared library (`.so` file).

You need to compile this library with the `-fPIC` and `-fno-gnu-unique` flags.

Your library should export a function named `expose` with the following signature:

```
extern "C" arc::grph::IGraphic* expose(void);
```

This function will be called by the engine to create your game, and will serve as an entry point for your game.

Your game should also export a function named `unexpose` with the following signature:

```
extern "C" void unexpose(arc::grph::IGraphic* graphic);
```

It will be called as needed by the engine. It serves as a free function to free your library from the memory.

Your library also needs to export a function named `getType` with the following signature:

```
extern "C" arc::DLType getType(void);
```

It will be called by the engine to determine the type of your shared library.

In our case, the type will be `arc::DLType::GRAPHICAL`.

```
arc::DLType getType(void) {
    return arc::DLType::GRAPHICAL;
}
```

2 — Creating the library

In your graphical library, you should inherit the `arc::grph::Graphic` class to interact with the engine.

For basic logic, your graphical library must provide the following methods:

- `init`: called by the engine when the library is loaded.
- `destroy`: called by the engine when the library is unloaded.
- `isOpen`: called by the engine every frame to check if the window is open and the game should continue.
- `tick`: returns the time in seconds since the last frame (see Canvas and rendering below), for example 0.03 for fixed 30fps, but the usage of a clock is recommended.

For the rendering, you should implement the following methods that will be called by the game itself:

- `clear`: called to clear the screen.
- `render`: called to refresh the screen and display the content of the canvas.

There is also a `pollEvent` method that will be called by the game through the `arc::IManager::pollEvent` method (see Event handling section).

You must also provide the `loadCanvas` and `unloadCanvas` methods (see Canvas and rendering section).

The `registerSprite` is a method that will be called by the game to register a sprite. It is not implemented by any existing graphical library but is provided for convenience.

3 — Canvas and rendering

You need to create a class that inherits `arc::grph::Canvas` class that will be used to draw on the screen.

In your class, you should implement the following methods:

- `getCapacities`: returns the capacities of the canvas, the only capacity supported by the engine is `CanvasCapacity::BASIC`.
- `startDraw`: called by the game before drawing on the canvas.
- `endDraw`: called by the game after drawing on the canvas.
- `drawPoint`: called by the game to draw a square representing a single point on the canvas.
- `drawText`: called by the game to draw text on the canvas. The text will be drawn at the given position aligned to the grid.
- `drawSprite`: called by the game to draw a sprite on the canvas. The sprite will be drawn at the given position aligned to the grid. It is not implemented by any existing graphical library but is provided for convenience.

Typically, the window grid is composed of 40×30 points, and a point is 20×20 pixels, where the origin is located in the top left corner of the window. But depending on the implementation, this may not be the case.

Colors are represented through the `arc::grph::IColor` passed in argument, which provides the following methods:

- `getColorCode` to get the color code of the color, in the format `0xRRGGBB`.
- `getSymbol` to get the fallback symbol of the color, for libraries that does not support colors or terminals.

On your `arc::grph::Graphic`-inherited class, you will provide a way to load the canvas with the `loadCanvas` method, and to unload with the `unloadCanvas` method, which is typically:

```cpp
void MyGraphic::loadCanvas(std::shared_ptr<ICanvas>& canvas) {
    canvas.reset(new MyCanvas());
}

void MyGraphic::unloadCanvas(std::shared_ptr<ICanvas>& canvas) {
    canvas.reset();
}
```

The game will call the former when it needs to get a new canvas, and will call the latter when it needs to unload the created canvas.

With the canvas, the game will be able to draw with the `drawPoint`, `drawText` and `drawSprite` methods.

The game will call `startDraw` and `endDraw` when it needs to draw on the canvas.

4 — Event handling

Event handling is done by the engine through the `arc::IManager::pollEvent` method, that will call the `pollEvent` method of your graphical library.

```cpp
bool MyGraphic::pollEvent(arc::Event& event) {
    // Typically the event structure used by your implementation
    MyWindowEvent windowEvent;

    while (this->window.pollEvent(myWindowEvent)) {
        if (windowEvent.type == WINDOW_KEY_PRESSED) {
            event.type = arc::Event::KEYDOWN;
            event.keyboardInput.keyCode = ...:
            return true;
        }

        if (windowEvent.type == WINDOW_CLOSE) {
            event.type = arc::Event::QUIT;
            return true;
        }
        // ...
    }

    return false;
}
```

The key codes are specified in the `arc::KeyCode` enum, and matches the lowercase ASCII codes when possible. For example, `KeyCode::A` corresponds to `'a'` (with the key code 97) and `KeyCode::Z` corresponds to `'z'` (with the key code 122).