Implementation of new games:

In order to be able to implement new game libraries it's necessary to inherit from an interface called IGame and therefore implement all the member functions.

```
class IGame
{
protected:
    IGraphics *lib;
public:
    virtual void start_game() = 0;
    virtual std::string game_tick() = 0;
    virtual bool is_running() = 0;
    virtual void use_graphics_lib(IGraphics *lib) = 0;
    virtual ~IGame() = default;
    virtual void reload_all_objects() = 0;
};
```

Implementation of new graphics libraries:

As with implementation of new games, the graphic library you want to add, has to inherit from an interface. In this case it's called IGraphics.

```
class IGraphics
{
public:
    virtual void init_screen(int width, int height) = 0;
    virtual ~IGraphics() = default;
    virtual void *createObject(object_creation_data *object_data) = 0;
    virtual void draw(void *object, int x0, int x1, int y0, int y1) = 0;
    virtual void deleteObject(void *object) = 0;
    virtual std::string getPressedKey(void) = 0;
    virtual void destroy_screen() = 0;
    virtual void display_screen() = 0;
    virtual void clear_screen() = 0;
};
```

In your game library you have to work with the "object_creation_data" structure:

```
typedef struct {
    object_type type;
    std::string text;
    std::string path_to_resource;
    std::string color_name;
} object_creation_data;
```

You need to use this structure, as it is the argument you receive in your "createObject" function.

It holds the information you need in order to create, save and return the necessary graphical element to render. An important part of that structure is an enum called “object_type”:

```
enum object_type
{
   TEXT,
   SPRITE,
   RECT
};
```

We distinguish between texts, sprites (a graphical element typically loaded from a picture) and rects (rectangular forms).

Beside that you also need to include a special type of function in the header file of your game/graphic library:

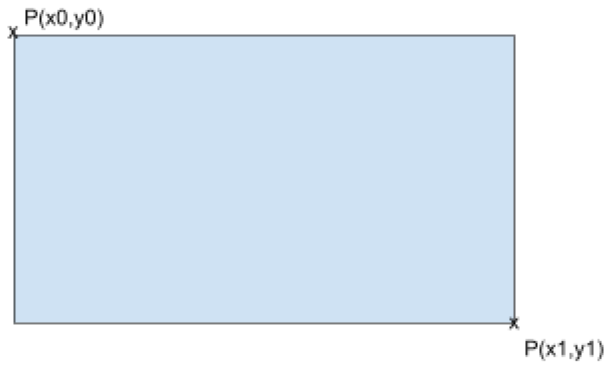
```
extern "C" {
  IGame *create_game(IGraphics *lib) {
     return new Nibbler(lib);  (This line is of course game/library specific)
  }
}
```

This function is needed, because C++ applies what is called name mangling (it adds extra parameters in the name). This causes problems when using the “dlsym” function to load our libraries. With the “extern “C”” declaration we use C linkage (no name mangling).

Other specifications one might not know just from seeing the interfaces:

Graphics:

- “draw” function:
  Beside an object pointer, it expects 4 arguments: “x0, x1, y0, y1”. Those are coordinates from which you can deduce not only the position of objects, but also their width and height.



- “getPressedKey” function:
  This function checks for key pressed events and return such event as a string:

  // on no user input: “”
  // on window exit: “exit”
  // on pressed enter: “enter”
  // on pressed space: “space”
  // on pressed backspace: “backspace”
  // on pressed escape: “esc”

// on pressed tabulator: “tab”
// ‘o’ → “prev_gr” (switch to previous library)
// ‘p’ → “next_gr” (switch to next library)
// ‘k’ → “prev_game” (switch to previous game)
// ‘l’ → “next_game” (switch to next game)
// all keys from a-z / 0-9 are recognized
// arrow keys are supported: “up”, “left”, “right”, “down”
// other input/ unsupported keys: “*”

Games:

- “game_tick” function:
  this function has to return event triggers as “esc”, which you received from the game libraries “getPressedKey” function

- “reload_all_objects” function:
  - for each object saved in the game lib (as void *) this, calls the lib->createObject function with the corresponding creation data.
  - Required to update all Object pointers when loading a new Graphics lib
  - to allow this function to work you must save the creation data of all objects created during the game within your game-lib

- saving usernames must be done in a file called “users” (each name on a separate line)
- saving highscores must be done in a file called “[game-name]_scores” e.g. “nibbler_scores” (e.g. “Username 3909”)