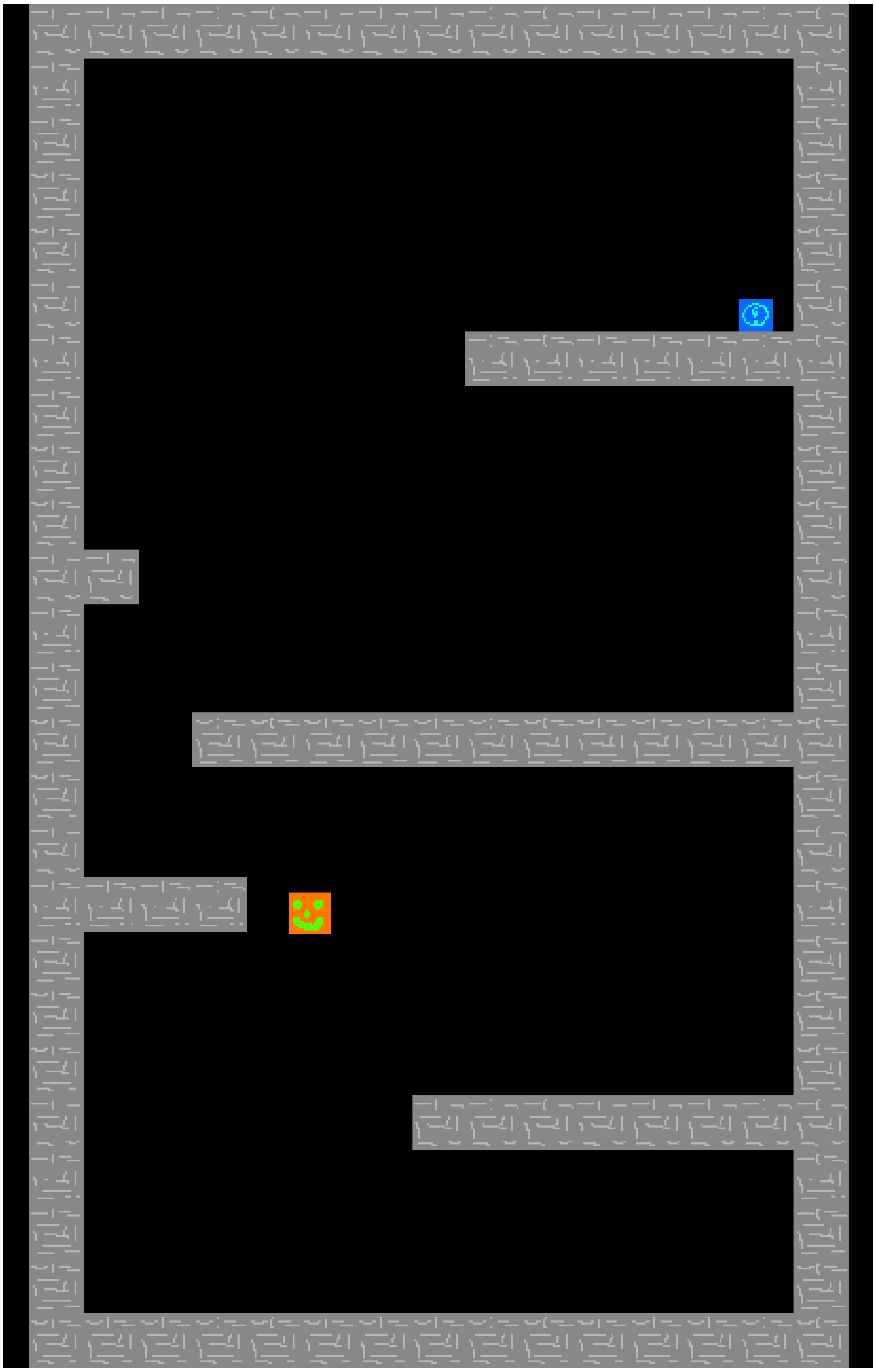
Meat-Boy project defense

Jonas Caluwé s0210051



Builder pattern

```
/// Game builder class used to configure `Game`.
class GameBuilder {
public:
    GameBuilder(WindowStyle style, std::pair size, const std::string&
title, bool vsync);

    /// Sets the fps if vsync is set to true.
    GameBuilder& set_fps(u32 fps);

    /// Returns a new instance of `Game` with the provided config.
    Game build() const;
private:
    const WindowStyle style;
    const std::pair size;
    const std::string title;
    const bool vsync;
    std::optional fps{};
};
```

A lot of systems require configuration that is often too complex to validate at compile time. This is where the “builder-pattern” is useful. It is found in the main game loop, level file parser and state manager.

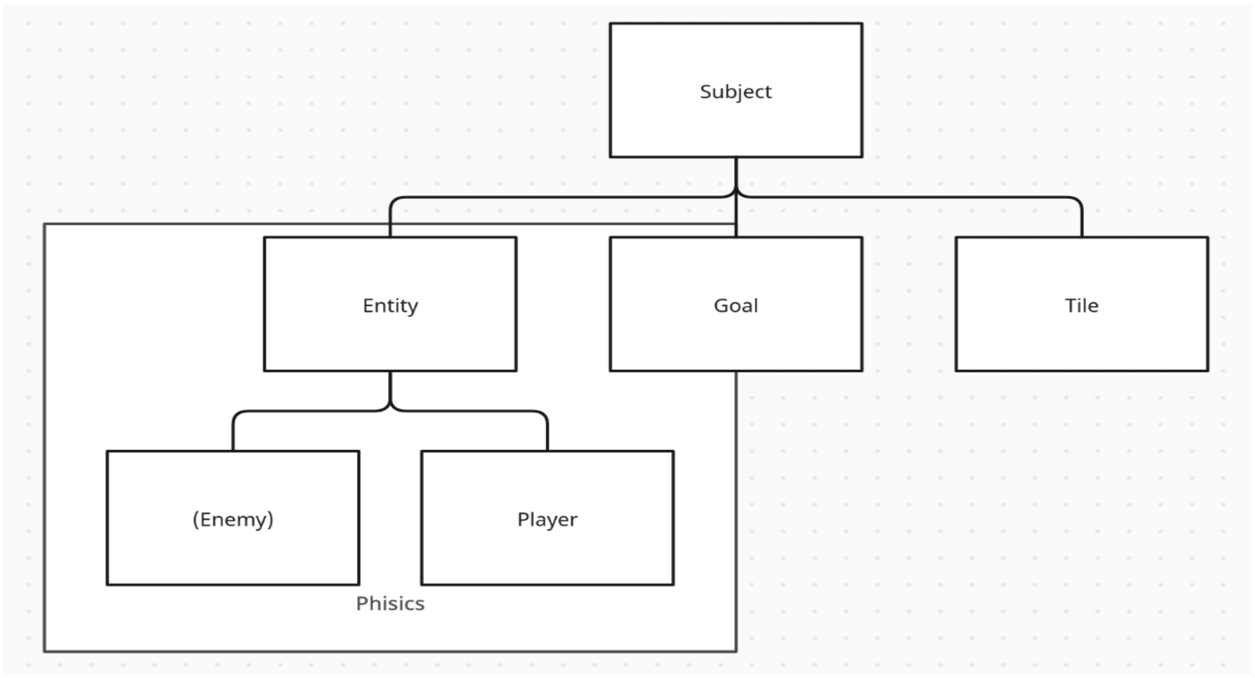
For the main game loop, I wanted the ability to configure fps, window title, whether to lock the frame rate, window style and size. When setting `vsync` to true, an fps value should be expected. This is checked in the `build` member function. When no fps value is set, a `runtime_error` is thrown. (Note, the `u32` type is an alias for `unsigned int`. More types like this are used throughout the project)

```
class LevelInfoBuilder {
public:
    inline LevelInfoBuilder(std::string name, u32 camera_height, f64
camera_increment);
    void set_size(const math::Vec2& size);
    void add_tile(const math::Vec2& tile);
    void set_goal(const math::Vec2& goal);
    LevelInfo build() const;
private:
    const std::string name{};
    std::optional<math::Vec2> size{};
    std::vector<math::Vec2> tiles{};
    std::optional<math::Vec2> player{};
    std::optional<math::Vec2> goal{};
    const u32 camera_height{};
    const f64 camera_increment{};
};
```

The level file parser needed the pattern not because of user configurability, but because guarantees like presence of a player and goal need to be kept.

The level file parser reads the level from top to bottom, this is however undesirable when working with coordinates. I did not want the user to specify the height of the level in the level file, because this is redundant information derivable from the level itself. I wanted to invert these coordinates to transform `(0, 0)` to the bottom left-most corner. This was nicely solvable using the `build` member function. It already knows about the size of the level and thus can invert the coordinates before setting them in the `LevelInfo` struct. This also allowed me to make the value members of the `LevelInfo` struct const.

Subject class hierarchy



At first I didn’t understand that the `Subject` class would be a marker class for anything observable. That is why, in the design I implemented, a `Subject` is anything in the world. It has a position, camera and bounding box/collider. How this impacts the design of the observer can be seen later. Objects that need physics to move or to interact with the world are `Entity``’s. Entities have velocity and acceleration. Entities collide only with tiles.

Observer pattern

As seen in the paragraph above, I have no class that marks a hierarchy as `Observable`. Instead, I chose to implement this using templates.

```
/// Abstract observer class
template <typename T>
class Observer {
public:
      virtual ~Observer() = default;

      /// Notify the observer of its task.
      virtual void notify(const T& source, ObserverEvent event) = 0;

protected:
      Observer() = default;
};
```

An `Observer` is anything that can be notified of an action. When it is notified, it is up to the observer to decide whether it is interested in the event and to extract the needed information from the object using its public interface. The template parameter specifies what the observer is looking at and what it can extract information from.

This design has a couple of drawbacks. First of all, every class that wants to be observable needs to implement and maintain a list of observers. Second, with the current design it is somewhat cumbersome to store observers, since the template parameter needs to match. This did not matter in this project, but is something that possibly limits the implementations of other types of projects.

State manager and resource manager

I again forgot to thoroughly read ahead when starting to program and did not realize that the `StateManager` should be in charge of changing levels. The current implementation uses a singleton `ResourceManager`. This keeps track of the parsed levels and the current active level.

Jump and wall detection

```
void Player::check_jump_collision(const std::vector<Bounds>& others) {
      // check bottom and sides using extended bound colliders, for both
sides and the bottom

      // amount to extend the collider with proportional to the current
size
      const f64 extend_amount_factor{0.1};
      const f64 extend_amount{extend_amount_factor *
this->get_rel_bounds().get_size().get_x()};

      const Bounds side_bounds(
      {this->get_abs_bounds().get_position().get_x() - extend_amount,
this->get_abs_bounds().get_position().get_y()},
      {this->get_rel_bounds().get_size().get_x() + extend_amount * 2,
this->get_rel_bounds().get_size().get_y()});

      const Bounds bottom_bounds(
      {this->get_abs_bounds().get_position().get_x(),
this->get_abs_bounds().get_position().get_y() - extend_amount},
      {this->get_rel_bounds().get_size().get_x(),
this->get_rel_bounds().get_size().get_y() + extend_amount});

      CollideInfo info{};
      for (const auto& other : others) {
      const bool side_collides = side_bounds.collides(other);
      const bool bottom_collides = bottom_bounds.collides(other);

      if (side_collides && this->get_abs_bounds().get_position().get_x() <
other.get_position().get_x())
            info.right = true;
      else if (side_collides)
            info.left = true;

      if (bottom_collides)
            info.down = true;
      }

      this->jump_collider = info;
}
```

Collision is the only part of the game that I would write differently today. When starting, I added a `project` member function to `Entity`. It updates averything physics related except the position. The position is checked inside the `World` and the correct position would be reported back. This worked fine until I needed jump detection. The latter is solved by first composing a list of all `Colliders` and giving these to the `Entity` so it can check whether it can jump or not. I should have done this for the regular collision detection as well.

Conclusion

Almost a quarter of the time working on Meat-Boy was spent on thinking about design and how systems would interact with each other. If I were to reimplement this project, I would mostly design it the same, other than the external collision detection. The graphics are a bit crude, but they should be fine as programmer art.