Ascii Maze

COLLABORATORS			
	TITLE : Ascii Maze		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Jordi Fita	February 6, 2018	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
b2fadabd1d1f	2011-08-22	Added the download links to Ascii Maze.	jfita
3211d6c8f6d7	2011-08-22	Imported 'Ascii Maze' from the ludumdare competition.	jfita

Contents

1 Game Play 1

2 The Maze Model 5

3 Maze Generators 12

3.1 Recursive Backtracking 15

3.2 Hunt-and-Kill 18

3.3 Binary Tree 21

3.4 Instantiating the Generators 22

4 Setting up the Maze 25

5 Game States 28

6 Entry Point 31

7 GUI Library 32

7.1 Widget Class 32

7.2 Label 33

7.3 Spinner 35

7.4 Select List 37

7.5 Tab Group 40

8 Building 42

9 License 46

10 References 47

10.1 References 47

List of Figures

1 Screenshot of *Ascii Maze* . . . 1
2 "Recursive Backtracker" starting position . . . 16
3 "Recursive Backtracker" making a path . . . 16
4 "Recursive Backtracker" goes back to a previous cell with unvisited neighbors and continues form there . . . 17
5 Call stack after calling `makePath(neighbor)` . . . 17
6 Call stack after returning from function . . . 17
7 "Hunt-and-Kill" starting position . . . 19
8 "Hunt-and-Kill" walking and creating a corridor . . . 19
9 "Hunting" for a new unvisited cell . . . 20
10 Selecting random east or south walls, except for the cells on the right border . . . 22
11 The binary tree generator has no choice on the last row . . . 22

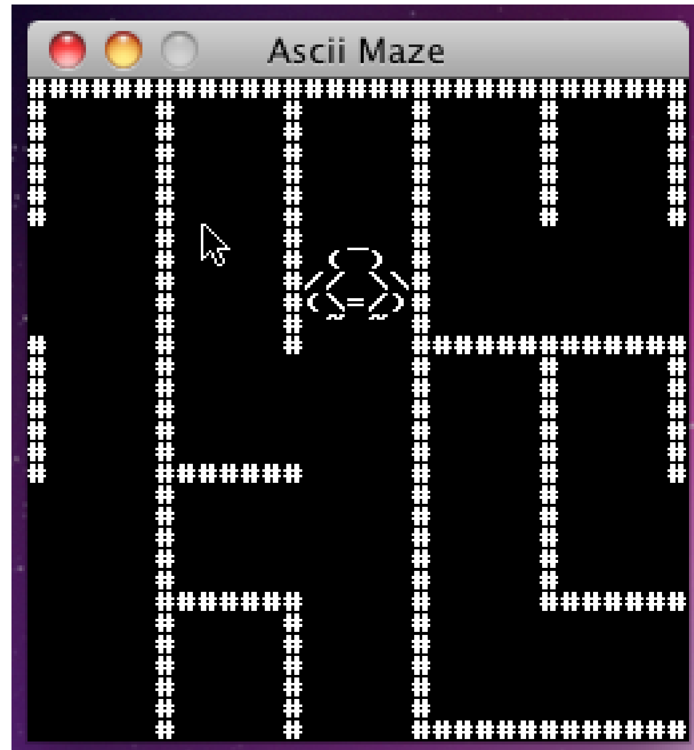

Figure 1: Screenshot of *Ascii Maze*

'Ascii Maze' is my entry for the 21st Ludum Dare 48H competition held on 19-21 August 2011. The theme for that competition was **Escape**.

'Ascii Maze' is a very simple, top-down maze game in which mazes are created using a set of selectable algorithms. When the game starts, it allows you to select a range of maze generation algorithms and also the board's size. Then, you move the character, an ugly penguin, through the generated maze and must find the exit to escape.

At the generator selection screen use the `up` and `down` cursor keys to select the algorithm to use to create the maze. Press `tab` to switch between the algorithm selection list and the board size spinner. To increase and decrease the board's size, hit `up` and `down` keys respectively, or the `Page Up` and `Page Down` to do increments in steps of 10 units. To generate the maze and start the game, press `Enter`. If you rather want to escape from the game, then find the `escape` key on your keyboard.

'Ascii Maze' uses *libtcod* as the rendering and input library and the Boost library. See [libtcod] and [Boost] for more information.

'Ascii Maze' was made using a software methodology called *Literate Programming* in which fragments of code are mixed between the prose in English explaining the game, as best as I can. These fragments are written in an order that makes more sense from the point of view of the reader rather than the order the compiler expects. In order to be able to compile the game, I need a *tangler* program that extracts these fragments and puts then in a form suitable to be understood by the compiler. The tangler program I am using is called *atangle* [atangle].

1 Game Play

The game play code for 'Ascii Maze' is actually quite straightforward and simple. All the game play is kept in a single class called `PlayState` which is an implementation of the `IGameState` interface described later in Section 5. For now it is enough

to know that this class is the responsible to move the player around the maze and to draw every element on the console; some sort of a controller and view at the same time. Thus, this class needs a `Maze` object, which acts as the game's model.

```
<<PlayState.hpp>>=
<<license>>
#if !defined(AM_PLAY_STATE_HPP)
#define AM_PLAY_STATE_HPP

#include "IGameState.hpp"
#include "Maze.hpp"

class PlayState: public IGameState
{
    public:
        <<PlayState constants>>

        PlayState(const Maze &maze);

        virtual void draw(TCODConsole &output);
        virtual void update(TCOD_key_t key, GameStateManager &stateManager);
        <<PlayState public functions declarations>>

    private:
        Maze maze_;
};

#endif // !AM_PLAY_STATE_HPP
```

```
<<PlayState.cpp>>=
<<license>>
#include "PlayState.hpp"
<<other includes of PlayState>>

PlayState::PlayState(const Maze &maze):
    maze_(maze)
{
    <<assert that the maze has the correct size>>
}

<<PlayState functions definitions>>
```

Every time the user wants to act on the maze, she always tells what through the `PlayState::update` function. The `update` expects a key with the command to perform and a `GameStateManager` to tell the game to move on to a different state, when appropriate. For example, to move the player around the maze, I only need to call the member functions of `Maze`.

```
<<PlayState functions definitions>>=
void
PlayState::update(TCOD_key_t key, GameStateManager &stateManager)
{
    if (key.pressed)
    {
        switch (key.vk)
        {
            case TCODK_DOWN:
                maze_.movePlayerDown();
                break;

            case TCODK_LEFT:
                maze_.movePlayerLeft();
                break;

            case TCODK_RIGHT:
```

```
                maze_.movePlayerRight();
                break;

            case TCODK_UP:
                maze_.movePlayerUp();
                break;

            <<other keys used while playing>>

            default:
                // Unknown key.  Nothing to do.
                break;
        }
    }
}
```

Another key that uses the `Maze` class as the model is when the player presses the `Enter` key. The `Enter` key is only used when the game is over — the player did escape the maze — to return to the main menu. Therefore, before I can go back to the main menu I need to check whether the player escaped or not. This is, again, an information I can retrieve from the model. (See Section 5 for an explanation of why I popping from the `stateManager` returns to the main menu.)

```
<<other includes of PlayState>>=
#include "GameStateManager.hpp"
```

```
<<other keys used while playing>>=
case TCODK_ENTER:
    if (maze_.didPlayerEscape())
    {
        stateManager.pop();
    }
    break;
```

Not all keys depend on the model, though. For instance, pressing `Escape` any time will end the game and will too return to the main menu.

```
<<other keys used while playing>>=
case TCODK_ESCAPE:
    stateManager.pop();
    break;
```

Drawing the maze and the player also retrieves every piece of information from the `Model`. The only thing that the `PlayState` class adds is scroll of the view, as the maze is usually way bigger than the console, and, of course, the actual screen representations for the cells and the player.

```
<<PlayState functions definitions>>=
void
PlayState::draw(TCODConsole &output)
{
    <<compute the view's scroll>>
    <<draw the visible cells>>
    <<draw the player>>
    <<draw the end of maze message>>
}
```

The scrolling is actually the trickiest part of drawing the maze, because I want to center the player on the screen, except for when the player moves to the maze's corners, where I want the view to be still and the player move towards the borders.

To center the player on the view, I need to know where the player is within the maze, something the maze can tell me, but I also need to know how many cells the view can show. I've decided to show only a 5x5 cells region of the maze.

```
<<PlayState constants>>=
static const int VIEW_WIDTH = 5;  // in cells
static const int VIEW_HEIGHT =  5; // in cells
```

Also, to make my life easier, the maze can't be smaller than the view region. That is, the view is always completely full.

```
<<assert that the maze has the correct size>>=
assert(maze_.width() >= VIEW_WIDTH);
assert(maze_.height() >= VIEW_HEIGHT);
```

The function that computes the scroll gets the player's position on the maze and then computes the first cell in X and the first cell in Y to start drawing on the console.

```
<<PlayState public functions declarations>>=
void computeViewStart(int *start_x, int *start_y);
```

```
<<compute the view's scroll>>=
int cell_start_x = 0;
int cell_start_y = 0;
computeViewStart(&cell_start_x, &cell_start_y);
```

Knowing from which cell start, the number of cells for each dimension, and that there can't be less cells that what is visible, to draw the cells I only need to make a for loop on the maze and draw each of the cells.

But, each cell's size isn't necessarily a character. Although it is of course possible to do, I like to have cells that are many characters in width and in height. This way I can draw pretty ASCII art as the player instead of using single character, as it is traditionally done in games such as Nethack.

Thus, each cell also has a width and a height, in characters, that I need to keep into account when drawing the cells.

```
<<PlayState constants>>=
static const int CELL_WIDTH = 6; // in characters
static const int CELL_HEIGHT = 6; // in characters
```

```
<<PlayState public functions declarations>>=
void draw(int x, int y, const Maze::Cell &cell, TCODConsole &output);
```

```
<<draw the visible cells>>=
for (int cell_y = 0, y = 0 ; cell_y < VIEW_HEIGHT ; ++cell_y, y += CELL_HEIGHT)
{
    for (int cell_x = 0, x = 0 ; cell_x < VIEW_WIDTH ; ++cell_x, x += CELL_WIDTH)
    {
        draw(x, y, maze_.cell(cell_start_x + cell_x, cell_start_y + cell_y),
            output);
    }
}
```

To draw the player, I only have to remember that I have to offset her position to center the player on the screen. That is, I have to take into account `cell_start_x` and `cell_start_y` here as well.

```
<<PlayState public functions declarations>>=
void draw(int offset_x, int offset_y, const Maze::Player &player, TCODConsole &output);
```

```
<<draw the player>>=
draw(cell_start_x, cell_start_y, maze_.player(), output);
```

2 The Maze Model

As previously state, the main game's model is the Maze class. This class has all the information to play and draw the game, but none of the interfaces to be controlled directly from the keyboard or to show anything on the console.

```
<<Maze.hpp>>=
<<license>>
#if !defined (AM_MAZE_HPP)
#define AM_MAZE_HPP

#include <boost/multi_array.hpp>

class Maze
{
    public:
        <<definition of the Cell struct>>

        <<grid type definition>>

        <<definition of the Player struct>>

        <<Maze constructor declaration>>

        <<Maze public functions declarations>>

    private:
        <<Maze private attributes>>
};

#endif // !AM_MAZE_HPP
```

```
<<Maze.cpp>>=
<<license>>
#include "Maze.hpp"
<<other Maze includes>>

<<Maze functions definitions>>
```

The `Maze` class is actually a grid of cells. A cell, in this context, is simply an structure that tells if it has a wall at each of the cardinal points: down, left, right, and left. Notice that as each cell has the information about their four directions, cells that share the same wall define the same wall twice. But this makes processing the maze a lot easier because when I look at a single cell, I know on which directions I can move and on which can't.

The `Cell` used by `Maze` is a `struct` instead of a `class` because this is a data only element. Encapsulating the access to the member attributes behind getters and setters won't be of any service. For convenience, a `Cell` can be initialized surrounded with walls or alternatively set each cell on the constructor.

```
<<definition of the Cell struct>>=
struct Cell
{
    bool bottom_wall;
    bool left_wall;
    bool right_wall;
    bool top_wall;

    Cell():
        bottom_wall(true),
        left_wall(true),
        right_wall(true),
        top_wall(true)
    {
```

```
    }

    Cell(bool bottom_wall, bool left_wall, bool right_wall, bool top_wall):
        bottom_wall(bottom_wall),
        left_wall(left_wall),
        right_wall(right_wall),
        top_wall(top_wall)
    {
    }
};
```

Drawing a cell is, then, a matter to check which walls must be drawn at border of the cell's size. But, the right and bottom walls are somewhat special because they are drawn *outside* the cell's border. This is to ensure that those cells on the maze's right and bottom borders get an outer wall as well. This is not a problem for inner cells, because the neighbour of a cell with a left wall has a right wall, and they overlap when drawing. That it, the player will only see a single wall even though I draw the same wall twice.

The only special case are the corners, because, for instance, if I tried to draw first the top as a wall and after I drew the right wall as an empty corridor, then the right wall would overwrite the top wall's corner and seams would appear. To prevent these seams, the corners are handled in a particular case in which I draw a corner if either converging wall is solid.

```
<<PlayState functions definitions>>=
void
PlayState::draw(int x, int y, const Maze::Cell &cell, TCODConsole &output)
{
    const int WALL_CHAR = '#';
    const int FLOOR_CHAR = ' ';

    int bottom_wall_char = cell.bottom_wall ? WALL_CHAR : FLOOR_CHAR;
    int left_wall_char = cell.left_wall ? WALL_CHAR : FLOOR_CHAR;
    int right_wall_char = cell.right_wall ? WALL_CHAR : FLOOR_CHAR;
    int top_wall_char = cell.top_wall ? WALL_CHAR : FLOOR_CHAR;

    // Top and bottom walls.
    for (int char_x = 1 ; char_x < CELL_WIDTH ; ++char_x)
    {
        output.putChar(x + char_x, y, top_wall_char);
        output.putChar(x + char_x, y + CELL_HEIGHT, bottom_wall_char);
    }
    // Left and right walls.
    for (int char_y = 1 ; char_y < CELL_HEIGHT ; ++char_y)
    {
        output.putChar(x, y + char_y, left_wall_char);
        output.putChar(x + CELL_WIDTH, y + char_y, right_wall_char);
    }

    // Middle
    for (int char_y = 1 ; char_y < CELL_HEIGHT ; ++char_y)
    {
        for (int char_x = 1 ; char_x < CELL_WIDTH ; ++char_x)
        {
            output.putChar(x + char_x, y + char_y, FLOOR_CHAR);
        }
    }

    // Corners
    if (top_wall_char == WALL_CHAR)
    {
        output.putChar(x, y, WALL_CHAR);
        output.putChar(x + CELL_WIDTH, y, WALL_CHAR);
    }
    if (bottom_wall_char == WALL_CHAR)
```

```
    {
        output.putChar(x, y + CELL_HEIGHT, WALL_CHAR);
        output.putChar(x + CELL_WIDTH, y + CELL_HEIGHT, WALL_CHAR);
    }
    if (left_wall_char == WALL_CHAR)
    {
        output.putChar(x, y, WALL_CHAR);
        output.putChar(x, y + CELL_HEIGHT, WALL_CHAR);
    }
    if (right_wall_char == WALL_CHAR)
    {
        output.putChar(x + CELL_WIDTH, y, WALL_CHAR);
        output.putChar(x + CELL_WIDTH, y + CELL_HEIGHT, WALL_CHAR);
    }
}
```

As said previously, `Maze` is a grid of `Cell` structures. However, the standard C++ library doesn't have any container that behaves as a dynamic array, thus I turn to Boost's `multi_array` to store the cells.

```
<<grid type definition>>=
typedef boost::multi_array<Cell, 2> grid_type;
```

```
<<Maze private attributes>>=
grid_type cells_;
```

This grid is initialized in the constructor by copying the data passed as parameter; another `grid_type` variable that needs to be squared and whose size can't be 0. That means that `Maze` is just a "container of cells and some logic" and that someone else needs to generate that data. See Section 3 to see the classes who generate the maze's cells.

```
<<Maze constructor declaration>>=
Maze(int player_x, int player_y, const grid_type &data);
```

```
<<other Maze includes>>=
#include <cassert>
```

```
<<Maze functions definitions>>=
Maze::Maze(int player_x, int player_y, const grid_type &data):
    cells_(data),
    <<other Maze initialization>>
{
    assert(width() == height());
    assert(height() > 0);
    <<other asserts for Maze's constructors>>
}
```

A public getter function in `Maze` is used to get a single cell from the grid, assuming the passed coordinate is valid.

```
<<Maze public functions declarations>>=
const Cell &cell(int x, int y) const;
```

```
<<Maze functions definitions>>=
const Maze::Cell &
Maze::cell(int x, int y) const
{
    assert(x < width());
    assert(y < height());
    return cells_[y][x];
}
```

Notice that this functions parameters — `x` and `y` — are reversed when accessing the actual grid. This seems confuse for no apparent reason, but it is due a limitation of my simple mind. I want to have the grid with the row in the first index and the column second because, usually, I'll loop the grid row first. Indexing this way is better cache wise. Unfortunately, when I have the x and the y coordinates "reversed" (i.e., first y followed by x) in a function call, I tend to forget it and use the more conventional coordinate order, resulting in stupid bugs that are even more confusing and, sometimes, harder to spot. I let the computer fix my shortcomings.

The `width` and `height` functions used in this function and the constructor are straightforward and leverage all the work to `multi_array`.

```
<<Maze public functions declarations>>=
int height() const;
int width() const;
```

```
<<Maze functions definitions>>=
int
Maze::height() const
{
    return cells_.shape()[0];
}

int
Maze::width() const
{
    return cells_.shape()[1];
}
```

Besides the grid's data, the initial position of the player is also defined in the constructor. This position is also stored inside an structure, for the same reason `Cell` is a structure, that not only has the current player's position, but also the direction it is facing. The maze doesn't actually need this but it is very useful when drawing the player on the screen in different positions depending on where it moves to.

Again, the constructor is just a helper function to initialize the whole class in a single call.

```
<<definition of the Player struct>>=
struct Player
{
    int x;
    int y;
    enum {
        DOWN = 0,
        LEFT = 1,
        RIGHT = 2,
        UP = 3
    } direction;

    Player(int x, int y):
        x(x),
        y(y),
        direction(UP)
    {
    }
};
```

```
<<Maze private attributes>>=
Player player_;
```

```
<<other Maze initialization>>=
player_(player_x, player_y)
```

```
<<other asserts for Maze's constructors>>=
assert(player_x < width());
assert(player_y < height());
```

Once set up, to draw the player I only need to get her current position and draw it according to the view's offset and the current direction. I can use the fact that the player's directions can be converted to integers to get the correct player's ASCII representation.

```
<<PlayState functions definitions>>=
void
PlayState::draw(int offset_x, int offset_y, const Maze::Player &player,
    TCODConsole &output)
{
    const char *player_faces[][5][CELL_HEIGHT] =
        {
            {
                "  _  ",
                " (v) ",
                "//-\\\\",
                "(\\_/)",
                " ~ ~ ",
                "     "
            },
            {
                "     ",
                " <') ",
                " /V\\ ",
                " (_)>",
                " ~~  ",
                "     "
            },
            {
                "     ",
                " ('> ",
                " /V\\ ",
                "<(_) ",
                "  ~~ ",
                "     "
            },
            {
                "  _  ",
                " ( ) ",
                "// \\\\",
                "(\\=/)",
                " ~ ~ ",
                "     "
            }
        };

    int output_x = (player.x - offset_x) * CELL_WIDTH + 1;
    int output_y = (player.y - offset_y) * CELL_HEIGHT;
    for (int y  = 1 ; y < CELL_HEIGHT ; ++y)
    {
        output.print(output_x, output_y + y,
            player_faces[player.direction][0][y - 1]);
    }
}
```

To get the current status of the player, `Maze` has a getter that returns the constant reference to this structure.

```
<<Maze public functions declarations>>=
```

```
const Player &player() const;
```

```
<<Maze functions definitions>>=
const Maze::Player &
Maze::player() const
{
    return player_;
}
```

This can be used to compute the scrolling for the view, as I now can access to the player's current position.

```
<<other includes of PlayState>>=
#include <algorithm>
```

```
<<PlayState functions definitions>>=
void
PlayState::computeViewStart(int *start_x, int *start_y)
{
    assert(start_x != NULL);
    assert(start_y != NULL);

    const Maze::Player &player = maze_.player();
    if (player.x < VIEW_WIDTH / 2)
    {
        *start_x = 0;
    }
    else
    {
        *start_x =
            std::min(maze_.width() - VIEW_WIDTH, player.x - VIEW_WIDTH / 2);
    }

    if (player.y < VIEW_HEIGHT / 2)
    {
        *start_y = 0;
    }
    else
    {
        *start_y =
            std::min(maze_.height() - VIEW_HEIGHT, player.y - VIEW_HEIGHT / 2);
    }
}
```

However, I can't move the player directly from outside the `Maze` class. I have to use its public member functions prepared for that. This is because I need to enforce that I don't try to move the player through a wall. Hence, before moving the player, I need to check whether in her cell there is a wall blocking the movement.

However, I always change the player's direction even if she can't move to the direction. Doing that, I give feedback to the player telling her that I understood the command, but I can't comply.

```
<<Maze public functions declarations>>=
void movePlayerDown();
void movePlayerLeft();
void movePlayerRight();
void movePlayerUp();
```

```
<<Maze functions definitions>>=
void
Maze::movePlayerDown()
{
    player_.direction = Player::DOWN;
```

```
    if ( !didPlayerEscape() && !cell(player_.x, player_.y).bottom_wall)
    {
        ++player_.y;
    }
}

void
Maze::movePlayerLeft()
{
    player_.direction = Player::LEFT;
    if ( !didPlayerEscape() && !cell(player_.x, player_.y).left_wall)
    {
        --player_.x;
    }
}

void
Maze::movePlayerRight()
{
    player_.direction = Player::RIGHT;
    if ( !didPlayerEscape() && !cell(player_.x, player_.y).right_wall)
    {
        ++player_.x;
    }
}

void
Maze::movePlayerUp()
{
    player_.direction = Player::UP;
    if ( !didPlayerEscape() && !cell(player_.x, player_.y).top_wall)
    {
        --player_.y;
    }
}
```

In each of these function I need to check whether the player has already escaped the maze or not. Not only because when the player escaped the game end and it is pointless to try to move the player, but also because when the player escapes she is no longer on any valid cell. If I tried to get the cell when she is outside the maze, I would have undefined behaviour. Why I don't prevent this by adding cells outside the maze, for instance? Because, as stated earlier, when the game is over I no longer need to move the player any further and I don't want to make the grid more complicated when I don't need it to be.

Moreover, this condition also makes easier to detect when the player escaped. If her position is "invalid", in the sense that doesn't belong to any cell in the grid, then she escaped.

```
<<Maze public functions declarations>>=
bool didPlayerEscape() const;
```

```
<<Maze functions definitions>>=
bool
Maze::didPlayerEscape() const
{
    return player_.x < 0 || player_.y < 0 ||
           player_.x >= width() || player_.y >= height();
}
```

This function can be used to draw an end of game kind of message.

```
<<draw the end of maze message>>=
if (maze_.didPlayerEscape())
{
```

```
    output.printFrame(2, 4, 27, 9, true);
    output.print(5, 6, "You escaped the maze!");
    output.print(4, 10, "Press ENTER to continue");
}
```

3 Maze Generators

The maze generators are the classes responsible to generate the data for a `Maze` class. Each generator implements a different strategy when creating the grid of cells, but all of the has the same common interface so I can use any generator in generic algorithms.

The common interface the generators is called `IMazeGenerator` and has a single virtual method, besides the constructor, called `generator` that generate and returns the maze data given a number of columns and rows. There is nothing preventing anyone passing a different number for columns and rows, but I only generate square mazes because it is easier for me to draw them later.

The `IMazeGenereator` interface definition is the following.

```
<<IMazeGenerator.hpp>>=
<<license>>
#if !defined(AM_INTERFACE_MAZE_GENERATOR_HPP)
#define AM_INTERFACE_MAZE_GENERATOR_HPP

#include <boost/multi_array.hpp>
#include <boost/array.hpp>
#include "Maze.hpp"

class IMazeGenerator
{
    public:

        virtual ~IMazeGenerator();

        virtual Maze::grid_type generate(int width, int height) = 0;

    protected:
        <<cell index type definition>>
        <<visited array definition>>

        <<generator utility functions declarations>>
};

#endif // !AM_INTERFACE_MAZE_GENERATOR_HPP
```

```
<<IMazeGenerator.cpp>>=
<<license>>
#include "IMazeGenerator.hpp"
<<other IMazeGenerator includes>>

IMazeGenerator::~IMazeGenerator()
{
}

<<IMazeGenerator functions definitions>>
```

I also need some common functions and type definitions that most of the generators will use. One of these is a function that makes as passage between two adjacent cells. This function expects the reference to the data with the cells to modify as well as the index to the two cells to joint. Then, looking at the relative position of these cells positions, the functions removes the walls so as the two cells are now merged.

```
<<cell index type definition>>=
typedef boost::array<Maze::grid_type::index, 2> cell_index;
```

```
<<generator utility functions declarations>>=
static void makePassage(Maze::grid_type &grid, const cell_index &cell1,
    const cell_index &cell2);
```

```
<<IMazeGenerator functions definitions>>=
void
IMazeGenerator::makePassage(Maze::grid_type &grid, const cell_index &cell1,
    const cell_index &cell2)
{
    if (cell1[0] == cell2[0])
    {
        if (cell1[1] < cell2[1])
        {
            grid(cell1).right_wall = false;
            grid(cell2).left_wall = false;
        }
        else if (cell1[1] > cell2[1])
        {
            grid(cell1).left_wall = false;
            grid(cell2).right_wall = false;
        }
    }
    else if (cell1[1] == cell2[1])
    {
        if (cell1[0] < cell2[0])
        {
            grid(cell1).bottom_wall = false;
            grid(cell2).top_wall = false;
        }
        else if (cell1[0] > cell2[0])
        {
            grid(cell1).top_wall = false;
            grid(cell2).bottom_wall = false;
        }
    }
}
```

Of course, the generators also have to add a random exit to the maze. It is possible to add the exit outside the generator, for instance the caller could add it, but I don't like the idea of making a new copy of the data when creating the maze. The copy I am referring to is the copy I would need to do from the generator's result, add the exit, and then copy again the data to the new maze. Like this:

```
Maze::grid_type data = generator->generate(5, 5); // copy
addExit(data);
Maze maze(1, 1, data); // copy
```

If I pass the result of the generator's result as a parameter to Maze's constructor, given that the parameter is a `const` reference, the compiler **could** perform a *return value optimization* [Meyers96] and avoid temporary copy. There is no guarantees though.

To add a maze's exit I simply select one of the four borders and then randomly pick a cell and remove the corresponding wall to allow the player exit the room.

```
<<generator utility functions declarations>>=
void addMazeExit(Maze::grid_type &cells);
```

```
<<IMazeGenerator functions definitions>>=
void
```

```
IMazeGenerator::addMazeExit(Maze::grid_type &cells)
{
    switch (rand() % 4)
    {
        // bottom
        case 0:
            cells[cells.shape()[0] - 1][rand() % cells.shape()[1]].bottom_wall = false;
            break;

        // left
        case 1:
            cells[rand() % cells.shape()[0]][0].left_wall = false;
            break;

        // right
        case 2:
            cells[rand() % cells.shape()[0]][cells.shape()[1] - 1].right_wall = false;
            break;

        // top
        case 3:
            cells[0][rand() % cells.shape()[1]].top_wall = false;
    }
}
```

Another commonly used functionality required for some generator is the ability to choose an unvisited cell's neighbor randomly. Of course, the maze generator interface can't know if a given cell is visited or not unless explicitly told. Thus, for these generators that need to retrieve a random neighbor, they also need to keep an array of booleans that tells whether a cell has been visited.

```
<<visited array definition>>=
typedef boost::multi_array<bool, 2> VisitedCells;
```

With the list of visited cells, now it is trivial to get a random neighbor: from the current cell's position, check if the four neighbors are visited or not and store them in an array if they aren't. Then, at the end, pick one of them at random.

```
<<generator utility functions declarations>>=
static cell_index getRandomNeighbor(const cell_index &cell, const VisitedCells &visited);
```

```
<<other IMazeGenerator includes>>=
#include <vector>
#include <cstdlib>
```

```
<<IMazeGenerator functions definitions>>=
IMazeGenerator::cell_index
IMazeGenerator::getRandomNeighbor(const cell_index &cell, const VisitedCells &visited)
{
    cell_index bottom = {{ -1, 0 }};
    cell_index left = {{ 0, -1 }};
    cell_index right = {{ 0, 1 }};
    cell_index up = {{ 1, 0 }};
    typedef boost::array<cell_index, 4> Directions;
    Directions directions = { bottom, left, right, up };

    std::vector<cell_index> neighbors;
    neighbors.reserve(4);

    for (Directions::iterator direction = directions.begin() ;
            direction != directions.end() ; ++direction)
    {
        cell_index neighbor = {{ cell[0] + (*direction)[0],
                                 cell[1] + (*direction)[1] }};
```

```
        if (neighbor[0] >= 0 && neighbor[0] < visited.shape()[0] &&
            neighbor[1] >= 0 && neighbor[1] < visited.shape()[1] &&
            !visited(neighbor))
        {
            neighbors.push_back(neighbor);
        }
    }

    if (neighbors.empty())
    {
        cell_index invalid_cell = {{ -1, -1}};
        return invalid_cell;
    }
    return neighbors[rand() % neighbors.size()];
}
```

For the `rand` function to return different values each time the application runs, I need to initialize the *seed* calling the `srand` function. In this case, I initialize the seed with the time the application started.

```
<<other main includes>>=
#include <ctime>
```

```
<<initialize the random number generator>>=
srand(time(0));
```

3.1 Recursive Backtracking

```
<<RecursiveBacktrackerGenerator.hpp>>=
<<license>>
#if !defined (AM_RECURSIVE_BACKTRACKER_GENERATOR_HPP)
#define AM_RECURSIVE_BACKTRACKER_GENERATOR_HPP

#include "IMazeGenerator.hpp"

class RecursiveBacktrackerGenerator: public IMazeGenerator
{
    public:
        virtual Maze::grid_type generate(int width, int height);

    private:
        <<RecursiveBacktracker private functions>>
};

#endif // !AM_RECURSIVE_BACKTRACKER_GENERATOR_HPP
```

The recursive backtracking is a depth first search algorithm that uses backtracking to go back when it find a cul-de-sac. The basic idea is to start with a random cell and mark it as visited. Then, choose a random unvisited neighbor cell to move to and remove the wall between the two cells. Then start again with this neighbor cell. When I find a cell that has no unvisited cells, then I backpedal until a cell that still has unvisited cells and continue from there.

```
<<RecursiveBacktrackerGenerator.cpp>>=
<<license>>
#include "RecursiveBacktrackerGenerator.hpp"
#include <cassert>

Maze::grid_type
RecursiveBacktrackerGenerator::generate(int width, int height)
{
    assert(width > 0);
```

```
    assert(height > 0);

    Maze::grid_type cells(boost::extents[height][width]);
    VisitedCells visited(boost::extents[height][width]);

    <<choose start cell for backtracker>>
    <<make path from start>>

    addMazeExit(cells);
    return cells;
}

<<other RecursiveBacktracker functions>>
```

The first thing I need to do is to choose a random cell to start the algorithm with.

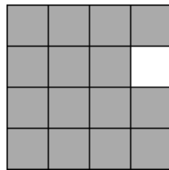

Figure 2: “Recursive Backtracker” starting position

```
<<choose start cell for backtracker>>=
cell_index start = {{ rand() % height, rand() % width }};
```

Then I start making a path from this cell.

```
<<RecursiveBacktracker private functions>>=
void makePath(const cell_index &node, VisitedCells &visited,
    Maze::grid_type &cells);
```

```
<<make path from start>>=
makePath(start, visited, cells);
```

To make a path, first I mark the node as visited and then I choose an unvisited neighbor randomly and make a passage between the two cells. Then I begin a new path calling the same `makePath` function recursively, but passing the neighbor cell, until I find a node that has no unvisited neighbors.

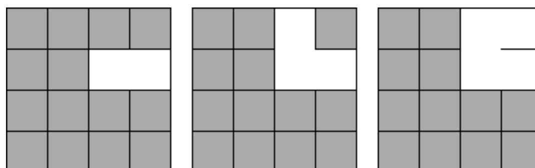

Figure 3: “Recursive Backtracker” making a path

```
<<other RecursiveBacktracker functions>>=
void
RecursiveBacktrackerGenerator::makePath(const cell_index &cell,
    VisitedCells &visited, Maze::grid_type &cells)
{
    assert(!visited(cell));
```

```
    visited(cell) = true;
    cell_index neighbor = getRandomNeighbor(cell, visited);
    while (neighbor[0] != -1)
    {
        makePassage(cells, cell, neighbor);
        makePath(neighbor, visited, cells);
        neighbor = getRandomNeighbor(cell, visited);
    }
}
```

When I find a cell that has all its neighbors visited I have to backtrack. That is, I must go back to a previous cell that still has unvisited neighbors and select one of these cells to continue making a new path.

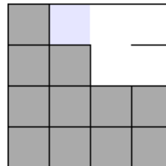

Figure 4: “Recursive Backtracker” goes back to a previous cell with unvisited neighbors and continues form there

Usually, to backtrack the algorithm should keep a stack of cells and add the visiting cells as soon as a `makePath` is called. Then, when I find a cell that can no longer make a pack, I would pop the previous cell from the stack and continue with it. However, as you surely have notice, I don’t keep any stack in the `RecursiveBacktrackerGenerator` object. That because C++ keeps a stack for me.

When I call `makePath` with the next neighbor cell as a parameter, C++ pushes the calls parameters to a stack as well as the caller return address in a *call stack*.

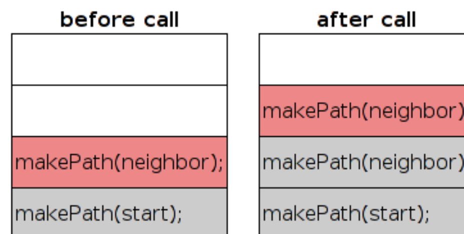

Figure 5: Call stack after calling `makePath(neighbor)`

When the function is over because I don’t have any more neighbors in the current cell to continue making a path, then I return and C++ *pops* the previous recursive call from the stack and continues after from where the call was made, which is the instruction to get the next neighbor.

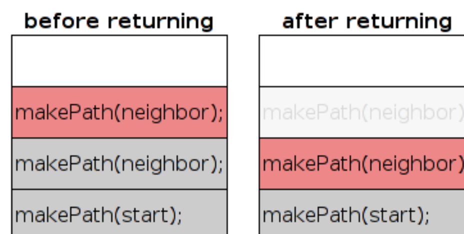

Figure 6: Call stack after returning from function

Thus, even if I don’t maintain an explicit stack for backtracking, C++ is doing the dirty work behind the scene.

3.2 Hunt-and-Kill

```
<<HuntAndKillGenerator.hpp>>=
<<license>>
#if !defined (AM_HUNT_AND_KILL_GENERATOR_HPP)
#define AM_HUNT_AND_KILL_GENERATOR_HPP

#include "IMazeGenerator.hpp"

class HuntAndKillGenerator: public IMazeGenerator
{
    public:
        virtual Maze::grid_type generate(int width, int height);

    private:
        <<HuntAndKill private functions>>
};

#endif // !AM_HUNT_AND_KILL_GENERATOR_HPP
```

The Hunt-and-Kill algorithm [Buck11-1] is an algorithm similar to a recursive backtracking: I start from a cell and randomly walk around the maze making passages to unvisited neighbors until I find a cell that has no unvisited neighbors. Then, instead of backtracking, I “hunt” for an unvisited cell adjacent to an already visited cell, and start walking randomly again from this new cell. This is repeated until there are no more unvisited cells. Due to the algorithm’s random walk, the resulting mazes tend to have long passages.

```
<<HuntAndKillGenerator.cpp>>=
<<license>>
#include "HuntAndKillGenerator.hpp"
#include <cassert>

Maze::grid_type
HuntAndKillGenerator::generate(int width, int height)
{
    assert(width > 0);
    assert(height > 0);

    Maze::grid_type cells(boost::extents[height][width]);
    VisitedCells visited(boost::extents[height][width]);

    size_t remaining_cells = width * height;
    <<choose a random hunt starting position>>
    while (remaining_cells > 0)
    {
        <<choose a random neighbor>>
        if (neighbor[0] == -1)
        {
            <<hunt for an unvisited neighbor>>
        }
        else
        {
            <<move to the neighbor>>
        }
    }

    addMazeExit(cells);
    return cells;
}

<<other HuntAndKillGenerator functions definitions>>
```

Once I've initialized the grid and the array of visited cells, setting each cell to an unvisited state, the first thing I do is randomly choose an starting position.

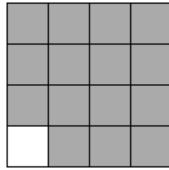

Figure 7: "Hunt-and-Kill" starting position

```
<<choose a random hunt starting position>>=
cell_index current = {{ rand() % height, rand () % width }};
visited(current) = true;
--remaining_cells;
```

Then I walk around the maze by picking unvisited neighbors of the current cell and carving a passage between the two cells. I do this until I find a cell that has no unvisited neighbors and the call to `getRandomNeighbor` returns an invalid cell with its coordinates set to -1.

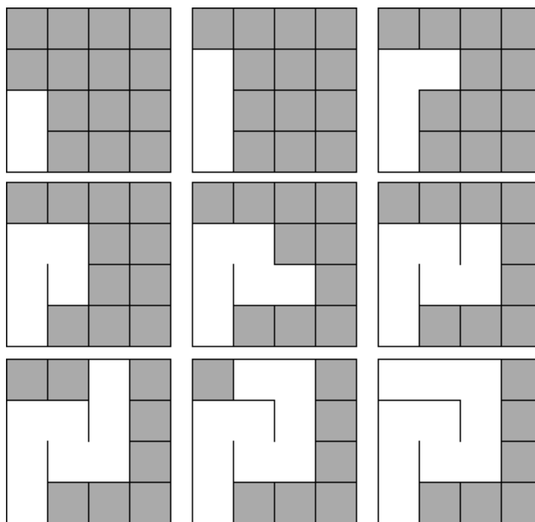

Figure 8: "Hunt-and-Kill" walking and creating a corridor

```
<<choose a random neighbor>>=
cell_index neighbor = getRandomNeighbor(current, visited);
```

```
<<move to the neighbor>>=
makePassage(cells, current, neighbor);
current = neighbor;
visited(current) = true;
--remaining_cells;
```

When I find a cell that has no unvisited cells, then it is time for "hunting" a new unvisited cell that is adjacent to a visited cell by looping through the entire maze. I make a passage between the unvisited and the visited and continue the algorithm from this unvisited cell.

A possible optimization would be to have a list of "completed rows" where I mark each row that has every node already visited and so I can skip this row when hunting. This optimization is not implemented here.

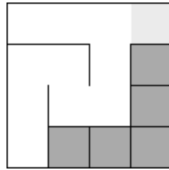

Figure 9: “Hunting” for a new unvisited cell

```
<<hunt for an unvisited neighbor>>=
current = hunt(visited, cells);
visited(current) = true;
--remaining_cells;
```

```
<<HuntAndKill private functions>>=
static cell_index hunt(const VisitedCells &visited, Maze::grid_type &cells);
```

```
<<other HuntAndKillGenerator functions definitions>>=
IMazeGenerator::cell_index
HuntAndKillGenerator::hunt(const VisitedCells &visited, Maze::grid_type &cells)
{
    cell_index bottom = {{ -1, 0 }};
    cell_index left = {{ 0, -1 }};
    cell_index right = {{ 0,  1 }};
    cell_index up = {{ 1, 0 }};

    typedef boost::array<cell_index, 4> Directions;
    Directions directions = {{ bottom, left, right, up }};

    int height = visited.shape()[0];
    int width = visited.shape()[1];
    cell_index cell;
    for (int row = 0 ; row < height ; ++row)
    {
        cell[0] = row;
        for (int col = 0 ; col < width ; ++col)
        {
            cell[1] = col;
            if (!visited(cell))
            {
                for (Directions::iterator direction = directions.begin() ;
                    direction != directions.end() ; ++direction)
                {
                    cell_index next = {{cell[0] + (*direction)[0],
                                        cell[1] + (*direction)[1] }};
                    if (next[0] >= 0 && next[0] < height &&
                        next[1] >= 0 && next[1] < width &&
                        visited(next))
                    {
                        makePassage(cells, cell, next);
                        return cell;
                    }
                }
            }
        }
    }
}
```

3.3 Binary Tree

```
<<BinaryTreeGenerator.hpp>>=
<<license>>
#if !defined (AM_BINARY_TREE_GENERATOR_HPP)
#define AM_BINARY_TREE_GENERATOR_HPP

#include "IMazeGenerator.hpp"

class BinaryTreeGenerator: public IMazeGenerator
{
    public:
        virtual Maze::grid_type generate(int width, int height);
};

#endif // !AM_BINARY_TREE_GENERATOR_HPP
```

The Binary Tree Generator constructs a maze following the same idea as if it was making a random binary tree [Buck11-2]. This algorithm also doesn't need to keep any additional data while building; it can build the entire maze with the information available at each cell.

The idea behind this algorithm is to remove from every cell a random wall from the sets of diagonal walls: north/east, north/west, south/east, and south/west. However, for every cell I need to use the same set. For this implementation I've decided to use the south and east walls.

The downside of this algorithm is that is has a bias of creating corridors towards the selected diagonal set — south/east, in this case — meaning that never will be a dead-end on that direction. This makes solving the maze a trivial task.

```
<<BinaryTreeGenerator.cpp>>=
<<license>>
#include "BinaryTreeGenerator.hpp"
#include <cassert>

Maze::grid_type
BinaryTreeGenerator::generate(int width, int height)
{
    assert(width > 0);
    assert(height > 0);

    Maze::grid_type cells(boost::extents[height][width]);

    cell_index cell;
    for (cell[0] = 0 ; cell[0] < height ; ++cell[0])
    {
        for (cell[1] = 0 ; cell[1] < width ; ++cell[1])
        {
            <<carve a wall either south or east>>
        }
    }

    addMazeExit(cells);
    return cells;
}
```

The algorithm loops the maze sequentially starting from the top right cell, although it could use any other order it wanted, provided it visited all cells once.

For each cell it has to decide whether to carve a new corridor either to the south or toward the east. For most cells, the algorithm chooses one or the other randomly, but for the cells that are at the far east border it only can choose south, otherwise it would move outside the maze's border.

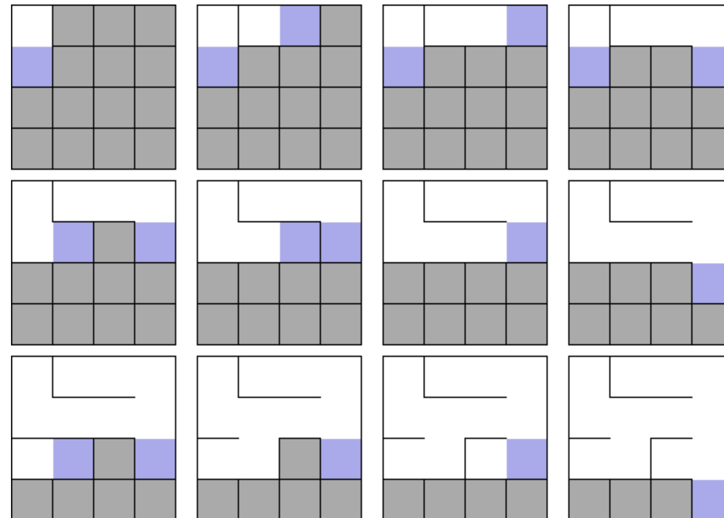

Figure 10: Selecting random east or south walls, except for the cells on the right border

The last row is special, because I can't never choose south lest I move outside the maze. And at the last cell I can't even do anything all; neither south nor east.

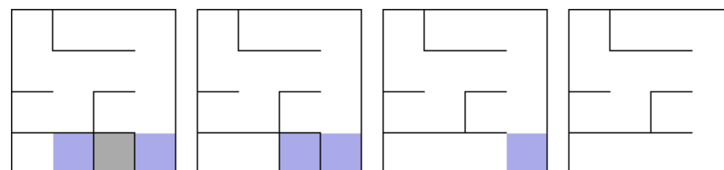

Figure 11: The binary tree generator has no choice on the last row

```
<<carve a wall either south or east>>=
if (cell[1] < width - 1)
{
    cell_index neighbor = {{cell[0], cell[1]}};
    int direction = (cell[0] < height - 1) ? rand() % 2 : 1;
    ++neighbor[direction];
    makePassage(cells, cell, neighbor);
}
else if (cell[0] < height - 1)
{
    cell_index neighbor = {{cell[0] + 1, cell[1]}};
    makePassage(cells, cell, neighbor);
}
```

3.4 Instantiating the Generators

In order to easier the instantiation of the different generators, I have a helper class that has the list of all available generators. This list has the generator name, the generator's description and the functor that instantiates new generator when asked to. This helper is called `GeneratorFactory`.

```
<<GeneratorFactory.hpp>>=
<<license>>
#if !defined(AM_GENERATOR_FACTORY_HPP)
#define AM_GENERATOR_FACTORY_HPP
```

```
<<GeneratorFactory headers>>

class GeneratorFactory
{
    public:
        <<definition of the generator information struct>>

        <<declaration of the function to initialize the list of generators>>
        <<declaration of the function to instantiate the generator>>
        <<declaration of the function to get the list of generators>>

    protected:
        <<make GeneratorFactory uncopiable>>

    private:
        <<the map of generators>>
};

#endif // AM_GENERATOR_FACTORY_HPP
```

This class consists only of `static` member functions and attributes, and thus don't need to be copied around. To prevent to copy it accidentally and to confuse people, I don't allow copies of this class by declaring the constructor, copy constructors, and assignment operator as protected and leaving out their definitions.

```
<<make GeneratorFactory uncopiable>>=
GeneratorFactory();
GeneratorFactory(const GeneratorFactory &);
GeneratorFactory &operator=(const GeneratorFactory &);
```

The most important type definition of this class is the `GeneratorInformation` structure. This structure contains the the generator description and the function object that creates new instances of this generator.

The name and the description are simply strings. The function object — the functor — is an instance of Boost's `function` class that allows me to store plain C function pointers in the same way as I could use actual classes with `operator()` defined.

```
<<GeneratorFactory headers>>=
#include <map>
#include <string>
#include <boost/shared_ptr.hpp>
#include <boost/function.hpp>
#include "IMazeGenerator.hpp"
```

```
<<definition of the generator information struct>>=
struct GeneratorInfo
{
    typedef boost::shared_ptr<IMazeGenerator> pointer_type;
    typedef boost::function<pointer_type ()> func_type;

    std::string description;
    func_type instantiator;

    GeneratorInfo(const std::string &description, const func_type &instantiator):
        description(description),
        instantiator(instantiator)
    {
    }
};

typedef std::map<std::string, GeneratorInfo> map_type;
```

The definition of all the generator is stored in a map with an string, the generator name, as the key.

```
<<the map of generators>>=
static map_type generators_;
```

```
<<GeneratorFactory.cpp>>=
<<license>>
#include "GeneratorFactory.hpp"
<<other GeneratorFactory includes>>

GeneratorFactory::map_type GeneratorFactory::generators_;

<<Instantiator template definition>>

<<GeneratorFactory functions definitions>>
```

The map of generators is initialized by another static member function of `GeneratorFactory` that simply adds the appropriate structures. It creates a template structure to instantiate the generator.

```
<<declaration of the function to initialize the list of generators>>=
static void initGenerators();
```

```
<<Instantiator template definition>>=
template<class generator_type> class Instantiator
{
    public:
        GeneratorFactory::GeneratorInfo::pointer_type operator()() const
        {
            return GeneratorFactory::GeneratorInfo::pointer_type(new generator_type);
        }
};
```

```
<<other GeneratorFactory includes>>=
#include "RecursiveBacktrackerGenerator.hpp"
#include "HuntAndKillGenerator.hpp"
#include "BinaryTreeGenerator.hpp"
```

```
<<GeneratorFactory functions definitions>>=
void
GeneratorFactory::initGenerators()
{
    generators_.insert(map_type::value_type("backtrack",
        GeneratorInfo("Backtracker",
            Instantiator<RecursiveBacktrackerGenerator>())));
    generators_.insert(map_type::value_type("hunt",
        GeneratorInfo("Hunt-and-Kill",
            Instantiator<HuntAndKillGenerator>())));
    generators_.insert(map_type::value_type("binary",
        GeneratorInfo("Binary Tree",
            Instantiator<BinaryTreeGenerator>())));
}
```

This function must be called from somewhere else, preferably the in `main`.

```
<<other main includes>>=
#include "GeneratorFactory.hpp"
```

```
<<initialize the map of generators>>=
GeneratorFactory::initGenerators();
```

Once the map is in place, to instantiate a generator, the only thing that the caller needs to know is the generator name. `getGenerator` looks for the generator and returns the pointers, if it could be found. Otherwise, throws an exception.

```
<<declaration of the function to instantiate the generator>>=
static GeneratorInfo::pointer_type getGenerator(const std::string &name);
```

```
<<other GeneratorFactory includes>>=
#include <stdexcept>
```

```
<<GeneratorFactory functions definitions>>=
GeneratorFactory::GeneratorInfo::pointer_type
GeneratorFactory::getGenerator(const std::string &name)
{
    map_type::const_iterator generator (generators_.find(name));
    if (generator == generators_.end())
    {
        throw std::runtime_error(
            std::string("Generator `") + name + "' not found");
    }
    return generator->second.instantiator();
}
```

To know which generators are available, a simple function returns a constant reference to the list, so nobody else can modify it.

```
<<declaration of the function to get the list of generators>>=
static const map_type &listGenerators();
```

```
<<GeneratorFactory functions definitions>>=
const GeneratorFactory::map_type &
GeneratorFactory::listGenerators()
{
    return generators_;
}
```

4 Setting up the Maze

The player must be able to play with the different maze generator as well as setting a custom maze size. This way the player can choose how she wants to lose her way into the maze.

Running the game without parameters and shows the set up screen. The setup screen allows the user to select which generator algorithm and which board size. It uses the widgets from the GUI library defined in Section 7 to ask the player for the maze parameters.

Being a game state, the setup screen is a class derived from `IGameState` as described in Section 5 later.

```
<<MazeSetupState.hpp>>=
<<license>>
#if !defined (AM_MAZE_SETUP_STATE_HPP)
#define AM_MAZE_SETUP_STATE_HPP

#include "IGameState.hpp"
#include "Label.hpp"
#include "Spinner.hpp"
#include "Select.hpp"
#include "TabGroup.hpp"

class MazeSetupState: public IGameState
{
    public:
        MazeSetupState();
```

```
        virtual void draw(TCODConsole &output);
        virtual void update(TCOD_key_t key, GameStateManager &stateManager);

    protected:
        TabGroup controls_;
        Label title_;
        Spinner maze_size_;
        Select generators_;
        Label instructions1_;
        Label instructions2_;
        Label instructions3_;
        Label instructions4_;
        Label instructions5_;
        Label instructions6_;
};

#endif // !AM_MAZE_SETUP_STATE_HPP
```

```
<<MazeSetupState.cpp>>=
<<license>>
#include "MazeSetupState.hpp"
<<other MazeSetupState includes>>

MazeSetupState::MazeSetupState():
    controls_(),
    title_(0, 1, "Maze Setup"),
    maze_size_ (1, 9, 18, 5, 99, 25, "Maze Size:"),
    generators_(1, 3, 18, 5, "Generator:"),
    instructions1_(0, 13, "Select generator and size"),
    instructions2_(0, 14, "TAB to move"),
    instructions3_(0, 15, "UP and DOWN change values"),
    instructions4_(0, 16, "PAGEUP and PAGEDOWN is faster"),
    instructions5_(0, 17, "ESCAPE to quit"),
    instructions6_(0, 23, "ENTER to start")
{
    title_.center(TCODConsole::root->getHeight());
    instructions1_.center(TCODConsole::root->getHeight());
    instructions2_.center(TCODConsole::root->getHeight());
    instructions3_.center(TCODConsole::root->getHeight());
    instructions4_.center(TCODConsole::root->getHeight());
    instructions5_.center(TCODConsole::root->getHeight());
    instructions6_.center(TCODConsole::root->getHeight());

    <<add the generators to the selection list>>

    controls_.addControl(&title_);
    controls_.addControl(&generators_);
    controls_.addControl(&maze_size_);
    controls_.addControl(&instructions1_);
    controls_.addControl(&instructions2_);
    controls_.addControl(&instructions3_);
    controls_.addControl(&instructions4_);
    controls_.addControl(&instructions5_);
    controls_.addControl(&instructions6_);
}

<<MazeSetupState functions definition>>
```

To generate the list of generator that can be selected, `MazeSetupState` uses the helper class `GeneratorFactory` that has every generator in a map. This map contains the name to use to instantiate the generator as well as a short description.

```
<<other MazeSetupState includes>>=
#include "GeneratorFactory.hpp"
```

```
<<add the generators to the selection list>>=
const GeneratorFactory::map_type &generators (GeneratorFactory::listGenerators());
for (GeneratorFactory::map_type::const_iterator generator = generators.begin();
     generator != generators.end() ; ++generator)
{
    generators_.addChoice(generator->second.description, generator->first);
}
```

To draw the controls, I only need to forward the call to the tab group that has the all the controls in.

```
<<MazeSetupState functions definition>>=
void
MazeSetupState::draw(TCODConsole &output)
{
    controls_.draw(output, true);
}
```

To update, I only have to check to see if the player pressed the enter of the escape keys. If she pressed enter, then I start a new maze with the specified maze' size and generator. Escape quits the game. Any other key is forwarded to the tab group to handle.

```
<<other MazeSetupState includes>>=
#include "GameStateManager.hpp"
```

```
<<MazeSetupState functions definition>>=
void
MazeSetupState::update(TCOD_key_t key, GameStateManager &stateManager)
{
    switch(key.vk)
    {
        case TCODK_ENTER:
            <<start a new maze>>
            break;

        case TCODK_ESCAPE:
            stateManager.pop();
            break;

        default:
            controls_.update(key);
            break;
    }
}
```

To start a new level, first I have to retrieve the board size and the generator name from the controls. Then, having the maze dimensions, I am able to set the player's initial position. Finally, I use the generator name to create a new instance of the `PlayState` class with a new maze.

```
<<other MazeSetupState includes>>=
#include "PlayState.hpp"
```

```
<<start a new maze>>=
{
    int size = maze_size_.value();
    std::string generator = generators_.selected();

    int player_x = rand() % size;
    int player_y = rand() % size;
```

```
    stateManager.push(new PlayState(
        Maze(player_x, player_y,
            GeneratorFactory::getGenerator(generator)->generate(size, size))));
}
```

5 Game States

As with many games, I need to keep a kind of Finite State Machine (FSM) to represent the various stages that it needs to be in (e.g., main menu, options menu, playing, pause, etc.) There are a lot of ways to make state machines in C++, but for game states I tend to use a variation of the *State* pattern [GoF95] called *Stack-Based State Machines* [Boer05].

Under this design pattern, each of the different game states is written in a different class that is derived from a common state interface. Even though 'Ascii Maze' I only have two different states — the setup and play states — I find this architecture convenient because every state needs to do practically the same tasks: flush the console, wait for a key, update the logic, draw to the console, and check whether the user closed the window to end the game. As such, I can move everything that is not specific for the state to the game loop itself, turning the game loop in a kind of *Template Method* pattern [GoF95].

```
<<game loop>>=
<<while there are active states>>
{
    <<get the current active state>>
    TCODConsole::root->clear();
    state->draw(*(TCODConsole::root));
    TCODConsole::flush();
    TCOD_key_t key = TCODConsole::waitForKeypress(true);
    state->update(key, stateManager);

    if (TCODConsole::isWindowClosed())
    {
        <<remove all active states>>
    }
}
```

Thus, every state now only needs to do two things: update its logic and draw to passed terminal reference. These two jobs are the functions that this common interface between the different states needs to have.

```
<<IGameState.hpp>>=
<<license>>
#if !defined (AM_INTERFACE_GAME_STATE_HPP)
#define AM_INTERFACE_GAME_STATE_HPP

#include <libtcod.hpp>

class GameStateManager;

class IGameState
{
    public:
        virtual ~IGameState() { }

        virtual void draw(TCODConsole &output) = 0;
        virtual void update(TCOD_key_t key, GameStateManager &stateManger) = 0;
};

#endif // !AM_INTERFACE_GAME_STATE_HPP
```

All games states are kept in a stack. The purpose of this stack is to be able to go back to the previous state without knowing which state it was. For instance, the main menu state will push to the stack a new playing state when appropriate, thus making

this new state the *active state*. When the game is over, instead of creating a new menu state and push it to the top of the stack, the playing state will just *pop* itself from the stack and therefore making the previous state (i.e., the menu) the active. With this logic, I can change the menus as much as I want (e.g., adding settings menus, credits, etc.) and the playing state is none the wise.

However, this stack needs to be a little more smart than a plain old stack. Mainly because when a state removes itself from the stack, I am still running the state's `update` member function. Removing an object when running one of its functions is not a great idea. I need a class, then, to manage two structures: the stack of active states and a list of states to be removed. This class is the *GameStateManger*

```
<<GameStateManager.hpp>>=
<<license>>
#if !defined(AM_GAME_STATE_MANAGER_HPP)
#define AM_GAME_STATE_MANAGER_HPP

#include <list>
#include <stack>
#include <boost/shared_ptr.hpp>
#include "IGameState.hpp"

class GameStateManager
{
    public:
        typedef boost::shared_ptr<IGameState> value_type;

        <<GameStateManager public declarations>>

    private:
        typedef std::stack<value_type> stack_type;
        typedef std::list<value_type> list_type;

        stack_type active_;
        mutable list_type inactive_;
};

#endif // !AM_GAME_STATE_MANAGER_HPP
```

```
<<GameStateManager.cpp>>=
<<license>>
#include "GameStateManager.hpp"
<<other GameStateManager includes>>

<<GameStateManager function definitions>>
```

Both the stack and the list use shared pointers because I don't want to bother to check when I need to release them. Once nobody, including the game state manger itself, holds a pointer to an state, `shared_ptr` deletes the state. Also, this means I don't need neither a custom constructor, copy constructor, destructor, or an assignment operator. The defaults just work.

I want to work with the `GameStateManager` just like a regular stack. Hence, I need functions to push, pop, and retrieve the top element of the manager. The `push` function is straightforward. Just remember to wrap the pointer with the smart pointer container. I don't want to pass the smart pointer directly as a parameter because then I can't push the result of a call to `new`.

```
<<GameStateManager public declarations>>=
void push(IGameState *state);
```

```
<<GameStateManager function definitions>>=
void
GameStateManager::push(IGameState *state)
{
    active_.push(value_type(state));
}
```

Popping the active state from the top of the stack is a little more involved, but not much. Basically, what I need to do is to first move the stack's top pointer to the list and then pop as usual.

```
<<GameStateManager public declarations>>=
void pop();
```

```
<<other GameStateManager includes>>=
#include <cassert>
```

```
<<GameStateManager function definitions>>=
void
GameStateManager::pop()
{
    assert(hasActiveState());
    inactive_.push_back(active_.top());
    active_.pop();
}
```

The `hasActiveState()` function simply tells if the stack has any element (i.e., is not empty)

```
<<GameStateManager public declarations>>=
bool hasActiveState() const;
```

```
<<GameStateManager function definitions>>=
bool
GameStateManager::hasActiveState() const
{
    return !active_.empty();
}
```

When there is no active state, then the game loop assumes that the game must quit and stops the loop.

```
<<while there are active states>>=
while (stateManager.hasActiveState())
```

This means that for the game to even start there must be at least a single state in the stack. For 'Ascii Maze', this state is the `MazeSetupState`, which allows the player to setup a new maze to play.

```
<<other main includes>>=
#include "MazeSetupState.hpp"
```

```
<<initialize the game state manager>>=
GameStateManager stateManager;
stateManager.push(new MazeSetupState);
```

This also means that to force the game to stop, I only need to remove all active states.

```
<<remove all active states>>=
stateManager.clear();
```

```
<<GameStateManager public declarations>>=
void clear();
```

```
<<GameStateManager function definitions>>=
void
GameStateManager::clear()
{
    while (hasActiveState())
    {
        pop();
    }
}
```

But when are deleted these inactive states? They are removed from the list, and thus deleted by `shared_ptr`, when the game loops requests the pointer to the currently active state. Keep in mind that the list of inactive states is just a way to avoid deleting objects at inappropriate moments, not a part of the interface nor the behaviour of the `GameStateManager`. When the game loops asks for the current active state what it means is “I am done with the current state now, what’s next?”. Therefore, this is the best moment to delete the list without having to a public function to `GameStateManager` to actively remove them. That is also why I’ve marked the list of inactive states as *mutable* and why `getActiveState` is *const*: I don’t change a bit the manager from the class user’s point of view.

```
<<get the current active state>>=
GameStateManager::value_type state = stateManager.getActiveState();
```

```
<<GameStateManager public declarations>>=
value_type getActiveState() const;
```

```
<<GameStateManager function definitions>>=
GameStateManager::value_type
GameStateManager::getActiveState() const
{
    assert(hasActiveState());
    inactive_.clear();
    return active_.top();
}
```

Notice how I didn’t use `getActiveState` in the implementation of `pop` even though both access to the same element (i.e., `active_.top()`.) This seemingly duplication of code was mode deliberately because I don’t want to delete inactive states when popping from the stack. The some state can pop more than a single state from the stack in a row (the first being itself) and deleting the inactive states would defeat the purpose of having the list on the first instance.

6 Entry Point

The application’s entry point is the responsible to initialize `libtcod` as well as any other possible resource that I can need while playing. Then I start the game loop.

The `main` function also has a general `try ... catch` block that catches any exception derived from the standard class as well as any other exception type. Here, I can’t do anything else than print the exception message, or an “unknown error” message if there isn’t any, and exit the application with an error code (i.e., `EXIT_FAILURE`.)

```
<<main.cpp>>=
<<license>>
#include <cstdlib>
#include <iostream>
#include <stdexcept>
#include "GameStateManager.hpp"
<<other main includes>>

int
main()
{
    try
    {
        <<initialize the random number generator>>
        <<initialize the map of generators>>
        <<initialize the root console>>
        <<initialize the game state manager>>
        <<game loop>>
        return EXIT_SUCCESS;
    }
    catch (std::exception &e)
    {
```

```
        std::cerr << e.what() << std::endl;
    }
    catch (...)
    {
        std::cerr << "Unknown error" << std::endl;
    }

    return EXIT_FAILURE;
}
```

To initialize the root console I just call `TCODConsole::initRoot` with the terminal exactly big enough to show the game's view, as defined in the constants for `PlayState` — `VIEW_WIDTH` and `VIEW_HEIGHT`, plus a character extra for the border. As these constants are expressed in characters, I don't need to convert anything.

The actual screen's resolution depends on this values as well as the font's size. The font I am using — called `terminal.png` because is the font name that libtcod loads by default — is a 16x16 pixels font.

Besides the terminal's size, libtcod also expects me to pass the window's title to show to the window manager. This is just for display purposes and of no consequence.

```
<<other main includes>>=
#include <libtcod.hpp>
#include "PlayState.hpp"
```

```
<<initialize the root console>>=
TCODConsole::initRoot(PlayState::VIEW_WIDTH * PlayState::CELL_WIDTH + 1,
    PlayState::VIEW_HEIGHT * PlayState::CELL_HEIGHT + 1, "Ascii Maze",
    false, TCOD_RENDERER_SDL);
```

7 GUI Library

Libtcod already provides a GUI library on top of the regular console functions, unfortunately, as far as I could see, this library's widget can only be controlled using the mouse. At least as far as keyboard focus is concerned. Also, the documentation of this library is near to non-existent and thus harder to figure out how to use it.

Since 'Ascii Maze' has very few requirements for GUI rendering, I made a GUI library similar to libtcod's but allowing the controls be manipulated with the keyboard, specially the tab control cycling.

7.1 Widget Class

The widget class is at the root of the controls hierarchy and defines the common interface that each control must support. Namely drawing themselves on a console and updating their values.

```
<<Widget.hpp>>=
<<license>>
#if !defined (AM_GUI_WIDGET_HPP)
#define AM_GUI_WIDGET_HPP

#include <libtcod.hpp>

class Widget
{
    public:
        Widget(int x, int y, int width, int height, bool tabStop);
        virtual ~Widget();

        virtual void draw(TCODConsole &output, bool hasFocus) = 0;
        virtual void update(TCOD_key_t key) = 0;
```

```
        <<Widget public functions>>

    private:
        bool tabStop_;
        int x_;
        int y_;
        int width_;
        int height_;
};

#endif // !AM_GUI_WIDGET_HPP
```

```
<<Widget.cpp>>=
<<license>>
#include "Widget.hpp"

Widget::Widget(int x, int y, int width, int height, bool tabStop):
    tabStop_(tabStop),
    x_(x),
    y_(y),
    width_(width),
    height_(height)
{
}

Widget::~Widget()
{
}
```

Most Widget's attributes are very common for a GUI library, such as the position — x and y — and the widget's dimensions. These can be changed with the Widget's getters and setters.

```
<<Widget public functions>>=
int x() const { return x_; }
int y() const { return y_; }
int width() const { return width_; }
int height() const { return height_; }
void setX(int x) { x_ = x; };
void setY(int y) { y_ = y; };
void setWidth(int width) { width_ = width; }
void setHeight(int height) { height_ = height_; }
```

The only attribute that could be confusing is the `tabStop_` variable, which significues whether the control can accept keyboard input or not. I.e., whether pressing the `tab` key should move the keyboard focus to the widget. This is a read only property.

```
<<Widget public functions>>=
bool hasTabStop() const { return tabStop_; }
```

The widget doesn't know whether it has the keyboard focus or now, this must be handled somewhere else. However, when drawing the widget, the caller must inform the widget whether it is focused or not so the widget can draw itself differently to give feedback to the user.

The `Widget` class is thus only a *data only* class and has no behaviour of its own. This must be added by the controls that subclass from this class.

7.2 Label

The label is the easiest of the controls because all it does is display an static string on the console. This text is stored inside the label object as an attribute.

```
<<Label.hpp>>=
<<license>>
#if !defined(AM_GUI_LABEL_HPP)
#define AM_GUI_LABEL_HPP

#include <string>
#include "Widget.hpp"

class Label: public Widget
{
    public:
        Label(int x, int y, const std::string &text);

        virtual void draw(TCODConsole &output, bool hasFocus);
        virtual void update(TCOD_key_t key);
        <<Label public functions>>

    private:
        std::string text_;
};

#endif // !AM_GUI_LABEL_HPP
```

The string passed as parameter to the constructor is also used to compute the control's width, which is the string's length. This control's height is always 1 and has no tab stop.

```
<<Label.cpp>>=
<<license>>
#include "Label.hpp"

Label::Label(int x, int y, const std::string &text):
    Widget(x, y, text.length(), 1, false),
    text_(text)
{
}

<<Label functions definitions>>
```

Being an static text, this widget has no behaviour and thus I need to leave the `update` function empty.

```
<<Label functions definitions>>=
void
Label::update(TCOD_key_t key)
{
    // Nothing to do.
}
```

Drawing the text is straightforward: I only need to print the string to the console on the specified position. There is no variation whether the control has focus or not.

```
<<Label functions definitions>>=
void
Label::draw(TCODConsole &console, bool hasFocus)
{
    console.print(x(), y(), text_.c_str());
}
```

Additionally, and for convenience, the `Label` class has also a function that centers the text on the given width. This can be used, for instance, to center a text on the screen.

```
<<Label public functions>>=
void center(int area);
```

```
<<Label functions definitions>>=
void
Label::center(int area)
{
    setX(area / 2 - width() / 2);
}
```

7.3 Spinner

The spinner is a control that allows the user to specify an integer within a given minimum and maximum. The spinner also contains a `Label` that is shown to the user at the spinner's left side.

```
<<Spinner.hpp>>=
<<license>>
#if !defined(AM_GUI_SPINNER_HPP)
#define AM_GUI_SPINNER_HPP

#include <string>
#include "Widget.hpp"
#include "Label.hpp"

class Spinner: public Widget
{
    public:
        Spinner(int x, int y, int width, int min, int max, int value,
            const std::string &label);

        virtual void draw(TCODConsole &output, bool hasFocus);
        virtual void update(TCOD_key_t key);
        <<Spinner public functions>>

    private:
        Label label_;
        int min_;
        int max_;
        int value_;
};

#endif // AM_GUI_SPINNER
```

When constructing the `Spinner` object, I must use the combined width of the label with the specified with for the spinner, plus an space. To know the label's width, I need to check again the string's length. The height is always 1 and this class can have the keyboard focus.

```
<<Spinner.cpp>>=
<<license>>
#include "Spinner.hpp"
<<other Spinner includes>>

Spinner::Spinner(int x, int y, int width, int min, int max, int value,
    const std::string &label):
    Widget(x, y, width + 1 + label.length(), 1, true),
    min_(min),
    max_(max),
    value_(value),
    label_(x, y, label)
{
}

<<Spinner functions definitions>>
```

To draw the spinner, first of all I need to draw the label at its position. Then I have to draw the value within a rectangle with a different background color. The width of the rectangle must be the widget's width less the label's width, because the width is the combined of these two.

Also, the background color for the rectangle depends on whether the control has the focus or not. When the control has focus I draw the rectangle with a blue background. When it is not focused, I draw a gray background.

```
<<Spinner functions definitions>>=
void
Spinner::draw(TCODConsole &output, bool hasFocus)
{
    label_.draw(output, false);
    TCODColor old_background = output.getDefaultBackground();
    if (hasFocus)
    {
        output.setDefaultBackground(TCODColor(40, 40, 120));
    }
    else
    {
        output.setDefaultBackground(TCODColor(70, 70, 70));
    }

    int margin = label_.width() + 1;
    output.printRectEx(x() + margin, y(), width() - margin, 1, TCOD_BKGND_SET,
        TCOD_LEFT, "%-*d", width() - margin, value_);
    output.setDefaultBackground(old_background);
}
```

To update the spinner's value based on the keyboard input, I assume that when this function is called means that the control has the keyboard. Here, I need to use the up and down arrow keys to increase or decrease the value respectively. Page Up and Page Down increase and decrease the value in steps of 10 instead. Of course, I must always check that the value does not moves out of limits using `std::min` and `std::max`.

```
<<other Spinner includes>>=
#include <algorithm>
```

```
<<Spinner functions definitions>>=
void
Spinner::update(TCOD_key_t key)
{
    switch(key.vk)
    {
        case TCODK_DOWN:
            value_ = std::max(value_ - 1, min_);
            break;

        case TCODK_UP:
            value_ = std::min(value_ + 1, max_);
            break;

        case TCODK_PAGEDOWN:
            value_ = std::max(value_ - 10, min_);
            break;

        case TCODK_PAGEUP:
            value_ = std::min(value_ + 10, max_);
            break;

        default:
            // Nothing else to do.
            break;
    }
```

```
}
```

Of course, this value must be accessible from outside to be useful.

```
<<Spinner public functions>>=
int value() const { return value_; }
```

7.4 Select List

The select list controls allows the user to select a single value from a multiple choices. The currently selected value is highlighted using a different color, which varies depending on whether the control has the keyboard focus. It can scroll the items to allow more options that can be displayed. The select list also has an attached label to display a prompt to the user.

The elements that can be selected are actually stored in a `std::vector` of `std::pair` containing two strings: one is the string that will be shown to the user and the other is the selected element's value. I use a `vector` instead of a `list` because I want to store the selected item as the index, to compute the correct scrolling when drawing. `vector` gives me random access to the elements, while `list` don't.

```
<<Select.hpp>>=
<<license>>
#if !defined (AM_GUI_SELECT_HPP)
#define AM_GUI_SELECT_HPP

#include <string>
#include <utility>
#include <vector>
#include "Label.hpp"
#include "Widget.hpp"

class Select: public Widget
{
    public:
        Select(int x, int y, int width, int height, const std::string &label);

        virtual void draw(TCODConsole &console, bool hasFocus);
        virtual void update(TCOD_key_t key);
        <<Select public functions>>

    private:
        typedef std::pair<std::string, std::string> value_type;
        typedef std::vector<value_type> list_type;

        list_type choices_;
        int selected_;
        Label label_;
};

#endif // AM_GUI_SELECT_HPP
```

Select's constructor gives the widget a width that is the sum of the specified width, where the choices will be shown, plus the label's width and a space of separation between them. The height is the number of choices that can be shown at the same time.

Initially, the selected choice will be -1, which means that no element is selected. This is because at this point I don't have any choice yet; they will get added at a later point. However, keep in main that no selection is **not** an error and thus the class user could want to have a list without any element, because it is waiting for some lengthy operations that will fill the list, for example.

```
<<Select.cpp>>=
<<license>>
#include "Select.hpp"
<<other Select headers>>
```

```
Select::Select(int x, int y, int width, int height, const std::string &label):
    Widget(x, y, width + 1 + label.length(), height, true),
    choices_(),
    selected_(-1),
    label_(x, y, label)
{
}

<<Select functions definitions>>
```

One of the first things any one would want to do with a Select is to add new choices. The `addChoice` member function simply adds the choice's value and display label inside the list as a `pair`. Also, if there is no selection made yet, it sets the selected item to be the first.

```
<<Select public functions>>=
void addChoice(const std::string &label, const std::string &value);
```

```
<<Select functions definitions>>=
void
Select::addChoice(const std::string &label, const std::string &value)
{
    choices_.push_back(std::make_pair(label, value));
    if (-1 == selected_)
    {
        selected_ = 0;
    }
}
```

Drawing this control means to draw the label and then each of the *visible* items.

```
<<Select functions definitions>>=
void
Select::draw(TCODConsole &output, bool hasFocus)
{
    label_.draw(output, false);
    TCODColor old_background = output.getDefaultBackground();
    <<set the colors to use for each choice type>>
    <<compute the list of visible choices>>
    <<draw the background rectangle>>
    <<draw the scroll bar if appropriate>>
    <<draw the visible choices>>
    output.setDefaultBackground(old_background);
}
```

This control also has variation of colors depending on whether it has the keyboard focus or not. However, in this case I have to set another background color for all the items, this in addition to the colors for both when the control has focus and when it doesn't. If I didn't do that, then moving the focus to a different control wouldn't display the selected item. So, I must set up the three colors, the control background and the unfocused and focused colors.

```
<<set the colors to use for each choice type>>=
TCODColor control_background = TCODColor(128, 120, 30);
TCODColor unfocused_choice = TCODColor(70, 70, 70);
TCODColor focused_choice = TCODColor(40, 40, 120);
```

I can have more choices that what can be shown in the control. Hence, I need to determine the range of elements that can need to be shown. Of course, the selected element must be the pivotal point when determining this range and must be visible all the time. Thus, the best way is to start the range assuming that the selected choice is the last element, but clamping to 0. To compute the end range, then I assume that can show as many choices as the control's height, beginning from the computed first element. This value is clamped to the choices' length to avoid an index out of bounds.

```
<<other Select headers>>=
#include <algorithm>
```

```
<<compute the list of visible choices>>=
int start = std::max(selected_ - height() + 1, 0);
int end = std::min(start + height(), static_cast<int>(choices_.size()));
```

Since most elements except for the selected will have the same background color, it seems natural to draw the whole control's background with the background color. Then, when necessary, I can override the background color when drawing the selected choice.

```
<<draw the background rectangle>>=
output.setDefaultBackground(control_background);
int control_start = x() + label_.width() + 1;
int control_width = width() - 1 - label_.width();
output.rect(control_start, y(), control_width, height(), true, TCOD_BKGND_SET);
```

I also want to let the user know when there are more choices that what is on the screen. For this, I will print an "scroll bar" on the control's side if I have more choices that room. This "scroll bar" is simply a up and down arrow. Having this scroll bar reduces the item's width by one.

```
<<draw the scroll bar if appropriate>>=
int item_width = control_width;
if (choices_.size() > height())
{
    --item_width;
    output.putChar(control_start + control_width - 1, y(), '^');
    output.putChar(control_start + control_width - 1, y() + height() - 1, 'v');
}
```

The last thing remaining to do is to draw the actual elements. I loop from the computed start to the end element in the choices list and print only the choice's label (the first value.) If the choice is selected, then I draw the item with either the focused or unfocused color depending on whether the control has the focus, while the rest must be drawn with the background color.

The list must only be drawn if there is any choice available, that is, if there is a selected element.

```
<<draw the visible choices>>=
if (selected_ > -1)
{
    for (int choice = start ; choice < end ; ++choice)
    {
        if (choice == selected_)
        {
            output.setDefaultBackground(
                hasFocus ? focused_choice : unfocused_choice);
        }
        else
        {
            output.setDefaultBackground(control_background);
        }
        output.printEx(control_start, y() + choice - start, TCOD_BKGND_SET,
                TCOD_LEFT, "%-*s", item_width, choices_[choice].first.c_str());
    }
}
```

Updating the control is way easier that drawing it. This control only reactions to the up and down cursor keys and I only have to increment and decrement the currently selected item keeping it within the choices' list limits. All this, if there is an already selected element, otherwise I don't have any choice in the list.

```
<<Select functions definitions>>=
void
Select::update(TCOD_key_t key)
{
    if (selected_ > -1)
    {
        switch (key.vk)
        {
            case TCODK_DOWN:
                selected_ = std::min(selected_ + 1,
                        static_cast<int>(choices_.size()) - 1);
                break;

            case TCODK_UP:
                selected_ = std::max(selected_ - 1, 0);
                break;
        }
    }
}
```

Finally, I can get the selected element from the list. If there is no selected element, I return the empty string.

```
<<Select public functions>>=
std::string selected() const;
```

```
<<Select functions definitions>>=
std::string
Select::selected() const
{
    if (selected_ > -1)
    {
        return choices_[selected_].second;
    }
    return "";
}
```

7.5 Tab Group

The Tab Group is an special control that is a just a group of other controls that must be selectable using the `tab` key. Basically, this is a list of control pointers and a single one of them has the keyboard focus.

```
<<TabGroup.hpp>>=
<<license>>
#if !defined (AM_GUI_TAB_GROUP_HPP)
#define AM_GUI_TAB_GROUP_HPP

#include <list>
#include "Widget.hpp"

class TabGroup: public Widget
{
    public:
        TabGroup();

        virtual void draw(TCODConsole &output, bool hasFocus);
        virtual void update(TCOD_key_t key);
        <<TabGroup public functions>>

    private:
```

```
        typedef Widget *value_type;
        typedef std::list<value_type> list_type;

        list_type controls_;
        list_type::iterator focused_;
};

#endif // !AM_GUI_TAB_GROUP_HPP
```

This class constructor's just pass to the widget some invalid values, because this control is invisible. The `focused_` attribute is set to the list's end because I don't have any control right now.

```
<<TabGroup.cpp>>=
<<license>>
#include "TabGroup.hpp"
<<other TabGroup includes>>

TabGroup::TabGroup():
    Widget(-1, -1, 0, 0, false),
    controls_(),
    focused_(controls_.end())
{
}

<<TabGroup functions definitions>>
```

To add new controls managed by this tab group, I have to add the *pointer* to a control using the `addControl` member function.

```
<<TabGroup public functions>>=
void addControl(Widget *control);
```

I opted to use pointers because is the only way I have to use the controls as a generic `Widget` object, because I can't copy references. Having said that, the `TabGroup` does not retain ownership of these pointers and thus it is not responsible to delete them. That is why this class doesn't need a destructor nor an user-defined copy constructor and assignment operator. The passed pointer must be a valid pointers; NULL pointers are not accepted.

```
<<other TabGroup includes>>=
#include <cassert>
```

```
<<TabGroup functions definitions>>=
void
TabGroup::addControl(Widget *control)
{
    assert(control != 0);
    list_type::iterator addedControl =
        controls_.insert(controls_.end(), control);
    <<set the initialy focused control>>
}
```

If the user adds a new control that has the tab stop property set to true and there is not yet any focused control, then tab group changes the iterator to the newly added control. This only works because I am using a `list` to store the controls and lists don't invalidate the pointers on insert. If I were using a `vector`, then I would have to use the actual pointers, and index, or something else, otherwise the application may crash when adding more controls after a widget with tab stop.

```
<<set the initialy focused control>>=
if (focused_ == controls_.end() && control->hasTabStop())
{
    focused_ = addedControl;
}
```

Drawing a tab control means drawing all the controls in the list telling to each one if they are the focused.

```
<<TabGroup functions definitions>>=
void
TabGroup::draw(TCODConsole &output, bool hasFocus)
{
    for(list_type::iterator control = controls_.begin() ;
        control != controls_.end() ; ++control)
    {
        (*control)->draw(output, control == focused_);
    }
}
```

Updating the tab group actually means updating the focused control. This means that there must be a focused at the time, otherwise would be impossible to update.

However, tab group also needs to keep an eye on the tab key, because in this case, instead of forwarding the key I need to select the next control that has tab stop. To select the next control, I move forward the `focused_` iterator until it is on a control with tab stop. If I reach the list's end, I start from the beginning again. This can't cause an endless loop because if I can update the tab group is because I had a control with tab stop to start with — the currently focused control. That means that the worse that can happen is that I end up selecting the same control again. Which is what I want.

```
<<TabGroup functions definitions>>=
void
TabGroup::update(TCOD_key_t key)
{
    assert(focused_ != controls_.end());

    if (key.vk == TCODK_TAB)
    {
        do
        {
            ++focused_;
            if (focused_ == controls_.end())
            {
                focused_ = controls_.begin();
            }
        }
        while (!(*focused_)->hasTabStop());
    }
    else
    {
        (*focused_)->update(key);
    }
}
```

8 Building

To build 'Ascii Maze' I am using *CMake* [cmake] as the build system. With CMake I can write the rules to build and link the application and other required targets with a simple declaration syntax and then CMake will convert these instructions to the platform's native build system (e.g., Makefiles, XCode®, Microsoft® Visual Studio® solution files.)

The instruction and rules that CMake requires in order to be able to build 'Ascii Maze' are written in a file called *CMakeLists.txt*. Since this game has a very simple directory structure, I only have a single CMakeLists.txt file in the project's root.

I start the CMakeLists.txt file telling which is the minimum required version of CMake that is needed in order to be able to read the file successfully. The syntax I use in this project is compatible with CMake's version 2.6 an forward.

```
<<CMakeLists.txt>>=
cmake_minimum_required(VERSION 2.6)
```

Then I tell CMake the project's name and the language it is written with. The project name is used as the name of some of output files, such as solution files, and the language is to be able to check whether the required compiler is installed on the system.

```
<<CMakeLists.txt>>=
project(LD21 CXX)
```

As I am using `libtcodxx`, I need to find first its include directory as well as its library file. If the include or library files are not found, then I can't build the game and must stop CMake.

```
<<CMakeLists.txt>>=

FIND_PATH(LIBTCOD_INCLUDE_DIR libtcod.hpp PATH_SUFFIXES libtcod
        PATHS /usr/include /usr/local/include /mingw/include)
IF(NOT LIBTCOD_INCLUDE_DIR)
    MESSAGE(FATAL_ERROR "Couldn't find libtcod include path")
ENDIF(NOT LIBTCOD_INCLUDE_DIR)

FIND_LIBRARY(LIBTCODXX_LIBRARIES NAMES tcodxx tcod-mingw tcod)
IF (NOT LIBTCODXX_LIBRARIES)
    MESSAGE(FATAL_ERROR "Couldn't find libtcodxx library")
ENDIF (NOT LIBTCODXX_LIBRARIES)
```

If I could find libtcod's include directory, then I tell CMake to use this path when compiling.

```
<<CMakeLists.txt>>=
include_directories(${LIBTCOD_INCLUDE_DIR})
```

I need to do the same for Boost, but this time CMake already provides the required modules to look for the include directories.

```
<<CMakeLists.txt>>=

find_package(Boost REQUIRED)
include_directories(${Boost_INCLUDE_DIR})
```

Being a literate game, in order to be able to build 'Ascii Maze', first I must extract and tangle the actual source code from the documentation. The best way to do that in CMake is using the `add_custom_command` command which adds a new rule that executes a series of commands to build a target when a series of dependences are modified.

Here, I am using 'cpif' to prevent changing the files' modification time for these sources that haven't actually changed their contents. See [cpif] for more information.

```
<<add the rule to tangle source code>>=
set(_document_full_path "${CMAKE_SOURCE_DIR}/README.txt")
add_custom_command(
    OUTPUT ${_output_full_path}
    COMMAND ${ATANGLE_BIN} -r "${_snippet}" ${_document_full_path} | ${CPIF_BIN} ${ ↩
        _output_full_path}
    MAIN_DEPENDENCY ${_document_full_path}
    COMMENT "Tangling source code ${output_file}")
```

The `ATANGLE_BIN` is the variable with the absolute path to the `atangle` application. Conversely, `CPIF_BIN` is the path to `cpif`. These variables are automatically set by CMake when creating the native build files by looking at the system's path, but the user can also specify an alternative path.

```
<<find literate tools>>=
find_program(ATANGLE_BIN NAMES atangle)
find_program(CPIF_BIN NAMES cpif)
```

However, typing the same commands again and again for each source file in the documentation soon becomes too repetitive and boresome. Also, `add_custom_command` does not ring a bell on whatever it is actually doing. Hence, it is better to move this command inside a macro that expects three parameters: the name of the output source file and the list to append the file to.

```
<<CMakeLists.txt>>=

<<find literate tools>>

macro(tangle_source output_file list_to_append)
    <<get the full path to the output file>>
    <<set the snippet name>>
    <<add the rule to tangle source code>>
    <<mark the output as a generated file>>
    <<append the output file to the list>>
endmacro(tangle_source)
```

CMake's `add_custom_command` requires both the full path to the output file as well as all its dependences. The full path of the dependence — this document — is well known. However, the full path to the source file depends on the directory in which this macro is called. The main idea is to generate this file in the project's *source* directory, not in the directory used to build, if it is different. This is because I want to distribute the game to people that don't have atangle installed. Thus, the output file's path is the current's source directory.

```
<<get the full path to the output file>>=
set(_output_full_path "${CMAKE_CURRENT_SOURCE_DIR}/${output_file}")
```

I also assume that the snippet name is the same as the output file's name.

```
<<set the snippet name>>=
set(_snippet ${output_file})
```

When I specify an executable or library target inside CMakeLists.txt, CMake tries to look for the source files to make sure that the build environment is sane. Unfortunately, for 'Ascii Maze' CMake won't actually find the source file until **after** the native build system generates them with the rule defined above in this macro. Then, I need to tell CMake that all the output files are *generated* and skip the test to check if they are available when running CMake.

```
<<mark the output as a generated file>>=
set_source_files_properties(${_output_full_path} PROPERTIES GENERATED ON)
```

The last thing this macro needs to do is to append the output file to the passed list. This list is simply another CMake variable that has every file needed to build the game. This is nothing that CMake requires me to do, but something I like to do to avoid duplicating the source file's names. If I have them in a list, I can pass the sources to `add_executable` later.

```
<<append the output file to the list>>=
list(APPEND ${list_to_append} ${_output_full_path})
```

With this macro set up, now I can list each source code and from which fragment to extract them from. I put the source files in two different lists: one for header files and the other for implementation files.

```
<<CMakeLists.txt>>=
set(LD21_SOURCES )
set(LD21_HEADERS )
set(LD21_RC )

tangle_source(am.rc LD21_RC)
tangle_source(GameStateManager.cpp LD21_SOURCES)
tangle_source(GameStateManager.hpp LD21_HEADERS)
tangle_source(BinaryTreeGenerator.cpp LD21_SOURCES)
tangle_source(BinaryTreeGenerator.hpp LD21_HEADERS)
tangle_source(GeneratorFactory.cpp LD21_SOURCES)
tangle_source(GeneratorFactory.hpp LD21_HEADERS)
tangle_source(HuntAndKillGenerator.cpp LD21_SOURCES)
tangle_source(HuntAndKillGenerator.hpp LD21_HEADERS)
tangle_source(IGameState.hpp LD21_HEADERS)
tangle_source(IMazeGenerator.hpp LD21_HEADERS)
```

```
tangle_source(IMazeGenerator.cpp LD21_SOURCES)
tangle_source(Label.cpp LD21_SOURCES)
tangle_source(Label.hpp LD21_HEADERS)
tangle_source(main.cpp LD21_SOURCES)
tangle_source(Maze.cpp LD21_SOURCES)
tangle_source(Maze.hpp LD21_HEADERS)
tangle_source(MazeSetupState.cpp LD21_SOURCES)
tangle_source(MazeSetupState.hpp LD21_HEADERS)
tangle_source(PlayState.cpp LD21_SOURCES)
tangle_source(PlayState.hpp LD21_HEADERS)
tangle_source(RecursiveBacktrackerGenerator.cpp LD21_SOURCES)
tangle_source(RecursiveBacktrackerGenerator.hpp LD21_HEADERS)
tangle_source(Select.cpp LD21_SOURCES)
tangle_source(Select.hpp LD21_HEADERS)
tangle_source(Spinner.cpp LD21_SOURCES)
tangle_source(Spinner.hpp LD21_HEADERS)
tangle_source(TabGroup.cpp LD21_SOURCES)
tangle_source(TabGroup.hpp LD21_HEADERS)
tangle_source(Widget.cpp LD21_SOURCES)
tangle_source(Widget.hpp LD21_HEADERS)
```

The reason I split the source files in header and implementation is because I then mark the header files as such and thus prevent the build system to try to compile them. Actually, CMake is smart enough to know the different between a header and an implementation file, but I like to mark them specifically anyway.

```
<<CMakeLists.txt>>=

set_source_files_properties(${LD21_HEADERS} PROPERTIES HEADER ON)
```

When building for Windows, I also want to build the resource files that adds metadata to the executable, such as version information and the icon.

The resource file is a text based file that must be compiled to a binary object, much like any other file, but the compiler is called *windres*.

Since CMake knows nothing about `windres`, I have to add another `add_custom_command` instruction. The input file is a file name `am.rc` and the output is a binary that I call `am.obj`. CMake also doesn't know how to deal with this binary file, and simply tries to use the file as-is when linking, hoping that this is what we want. And, in this case, it is.

Being a source, once the binary file gets generated, I append the output to the `LD21_SOURCES` list and also need to mark this file as generated to avoid nagging from CMake.

Notice that I output this file in the binary directory and not in the source directory as I do with the tangles source. That difference is because I don't plan to distribute the binary file with the rest of the source code.

```
<<CMakeLists.txt>>=

if(MINGW)
    set(LD21_RES ${CMAKE_CURRENT_BINARY_DIR}/am.obj)
    add_custom_command(OUTPUT ${LD21_RES}
        COMMAND windres.exe -I${CMAKE_SOURCE_DIR} -o ${LD21_RES} ${LD21_RC}
        MAIN_DEPENDENCY ${LD21_RC}
        COMMENT "Compiling resource file am.rc")
    list(APPEND LD21_SOURCES ${LD21_RES})
endif(MINGW)
```

The resource file, as I said, is just a text file with the game's metadata and the path to the icon file, which is located inside the `icons` folder at the project's root.

```
<<am.rc>>=
#include <windows.h>
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page (1252)
```

```
A ICON "icons/am.ico"

VS_VERSION_INFO VERSIONINFO
   FILEVERSION 1, 0, 0, 0
   PRODUCTVERSION 1, 0, 0, 0
   FILEFLAGSMASK VS_FFI_FILEFLAGSMASK
   FILEFLAGS VS_FF_PRERELEASE
   FILEOS VOS__WINDOWS32
   FILETYPE VFT_APP
   FILESUBTYPE VFT_UNKNOWN
   BEGIN
       BLOCK "StringFileInfo"
       BEGIN
           BLOCK "040904b0"
           BEGIN
               VALUE "CompanyName", "Geisha Studios"
               VALUE "FileDescription", "Simple top-down maze game"
               VALUE "FileVersion", "1.0.0.0"
               VALUE "InternalName", "ld21"
               VALUE "LegalCopyright", "Copyright (c) Jordi Fita"
               VALUE "OriginalFileName", "ld21.exe"
               VALUE "ProductName", "Ascii Maze"
               VALUE "ProductVersion", "1.0.0.0"
           END
       END
       BLOCK "VarFileInfo"
       BEGIN
           VALUE "Translation", 0x409, 1200
       END
   END
```

The only thing, then, that remains to do to build 'Ascii Maze' is to tell CMake that I want an executable using the tangled source code.

```
<<CMakeLists.txt>>=
add_executable(ld21 WIN32 ${LD21_HEADERS} ${LD21_SOURCES})
```

To link this executable, I need to use the libtcod I found earlier.

```
<<CMakeLists.txt>>=
target_link_libraries(ld21 ${LIBTCODXX_LIBRARIES})
```

9 License

'Ascii Maze' is released under the terms and conditions of the GNU General Public License [GPL] version 3.0 or, at your option, any later version. At the start of each source file there is the following notice.

```
<<license>>=
/*
 * Ascii Maze - A simple, top-view, auto-generated maze game.
 * Copyright (c) Jordi Fita <jfita@geishastudios.com>
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
```

```
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */
```


10 References

10.1 References

[1] [atangle] Jordi Fita. atangle - Geisha Studios. 2011. http://www.geishastudios.com/literate/atangle.html. [Online; accessed 20-August-2011].

[2] [Boer05] Jame Boer. Game Programming Gems 5: Large-Scale Stack-Based State Machines. 2005, Kim Palliser. ISBN 1-58450-352-1.

[3] [Boost] Boost C++ Libraries. 2011. http://www.boost.org/. [Online; accessed 20-August-2011].

[4] Jamis Buck. Maze Generation: Hunt-and-Kill algorithm. 2011. http://weblog.jamisbuck.org/2011/1/24/maze-generation-hunt-and-kill-algorithm. [Online; accessed 21-August-2011].

[5] [Buck11-2] Jamis Buck. Maze Generation: Binary Tree algorithm. 2011. http://weblog.jamisbuck.org/2011/2/-1/maze-generation-binary-tree-algorithm. [Online; accessed 21-August-2011].

[6] [cpif] Jordi Fita. cpif - Geisha Studios. 2011. http://www.geishastudios.com/literate/cpif.html. [Online; accessed 20-August-2011].

[7] [cmake] Kitware. CMake - Cross Platform Make. 2011. http://cmake.org/. [Online; accessed 20-August-2011].

[8] [GoF95] Gamma, et al. Design Patterns: Elements of Reusable Object-Oriented Software. 1995, Addison-Wesley. ISBN 0201633612.

[9] [GPL] Free Software Foundation. The GNU General Public License v3.0. 29 June 2007.

[10] [libtcod] Jice, Mingos, et al. The Dorye Library. 2010. http://doryen.eptalys.net/libtcod/. [Online; accessed 20-August-2011].

[11] [Meyers96] Scott Meyers. More Effective C++. 1996, Addison-Wesley. ISBN 020163371X.