An history of fundamental game algorithms with C++

Gaëtan Blaise-Cazalet - Head of programming section

Goals

What we will do

- ☑ Learn fundamental algorithms used in ANY virtual scene (games or cinema)
- ☑ Getting used to C++
- ☑ Apply good C++ habits (Efficient C++ / Efficient Modern C++, Scott Meyers)

What we won't do

- ☐ Design patterns. You can use them, but no focus on it : those games are simples. Keep complexity low. (But use OOP, please...)
- ☐ Optimization : "premature optimization is the root of all evil". Keep technicality low.
- ☐ Tutorials. You are programmers. You choose your implementation.

Resources

Tools

- SDL2 with sdl_image, sdl_mixer, sdl_ttf
- Either Visual Studio (C++ support)
- Or any light text editor with a Makefile and some batch. Like Visual Code.

SDL2 : https://www.libsdl.org/download-2.0.php

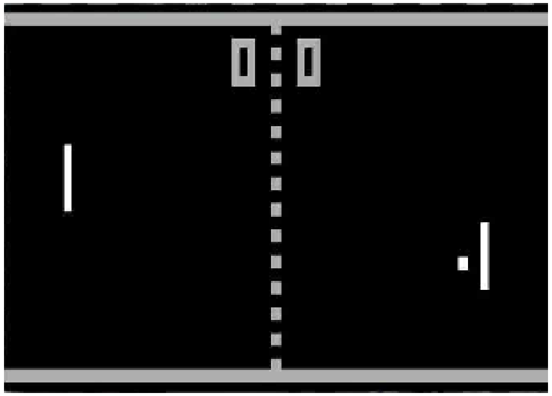
SDL2 image, mixer, ttf : https://www.libsdl.org/projects/

Quick start

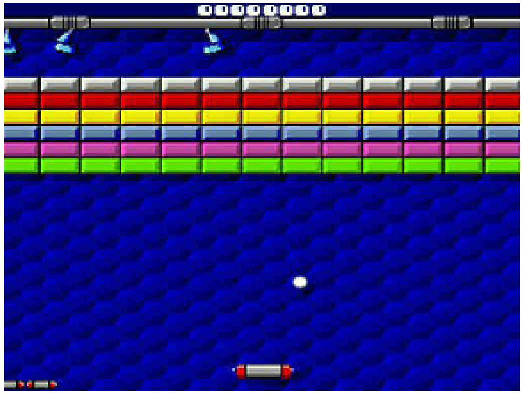
- A basic SDL2 project for Visual Code : https://github.com/Gaetz/sdl-Basic
- ...And some insight about config files (note it is meant to be compiled with g++, for an OpenGL project) : https://github.com/Gaetz/cpp-Tetris
- How to import sdl libs in big VS : http://lazyfoo.net/tutorials/SDL/

Menu

- Pong



- Brick Breaker



- Top-down Racer

- Tile Dungeon

- Space Shooter

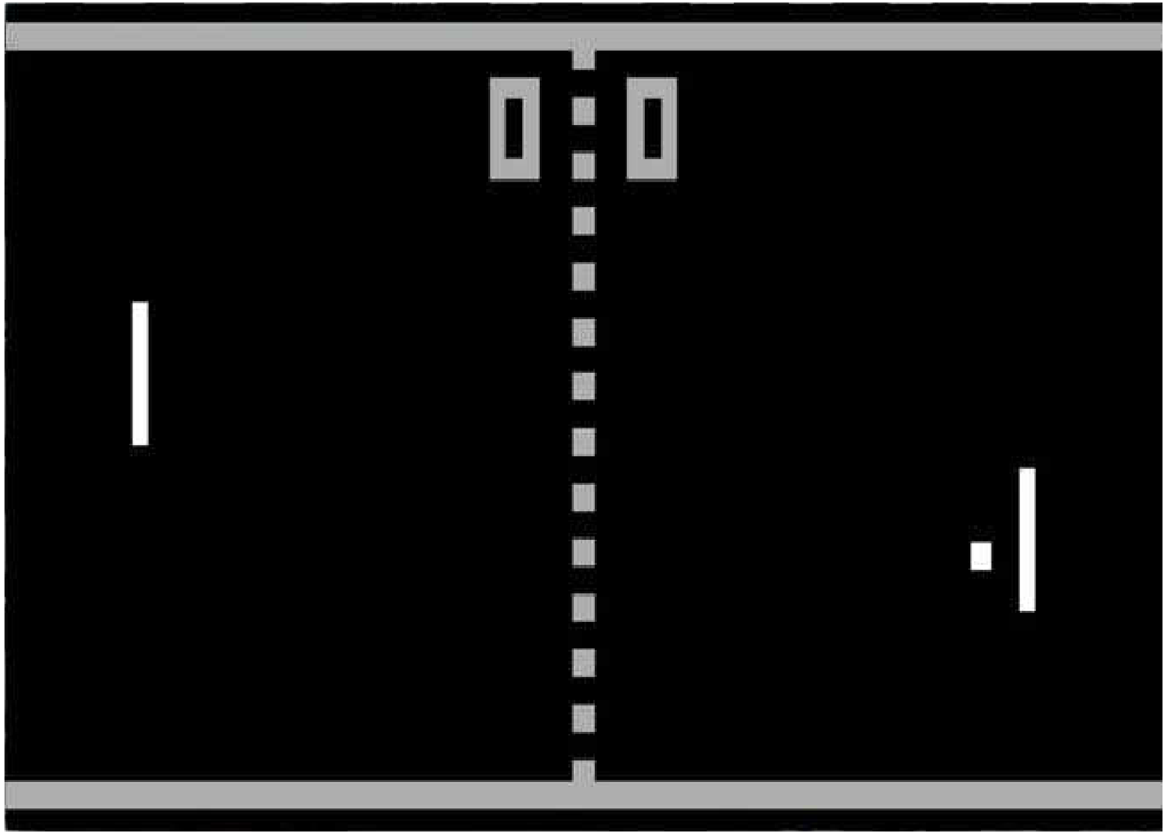
- 2D RTS

Two part for each project :

- First : programming classic algorithms
- Second : extend games to use more complex algorithms

First part - Classic algorithms

Pong



Window

- Create a 800 * 600 black window
- Create a classic empty game loop

Ball

- Draw a white ball
- (http://fvirtman.free.fr/recueil/02_03_04_formes.c.php)

- Don't spend time on optimization!

Ball move

- Make the ball move. To do so, update its coordinates with a speed.

The delta time is the delay since last frame display.

- Process the delta time
- Limit frame per seconds to 60 FPS
- Update the ball speed so it is multiplied by delta time

Ball bounce

- Make the ball bounce. Just inverse its speed when it reaches an edge of the screen.

Paddle

- Draw a paddle on the left edge of the screen.

Move Paddle

- Move paddle with the mouse. The paddle should be centered on the mouse.

Reset ball & Paddle bounce

- If the ball get out of screen on the left, reset its position
- Make the ball bounce on the left paddle

Opponent Paddle

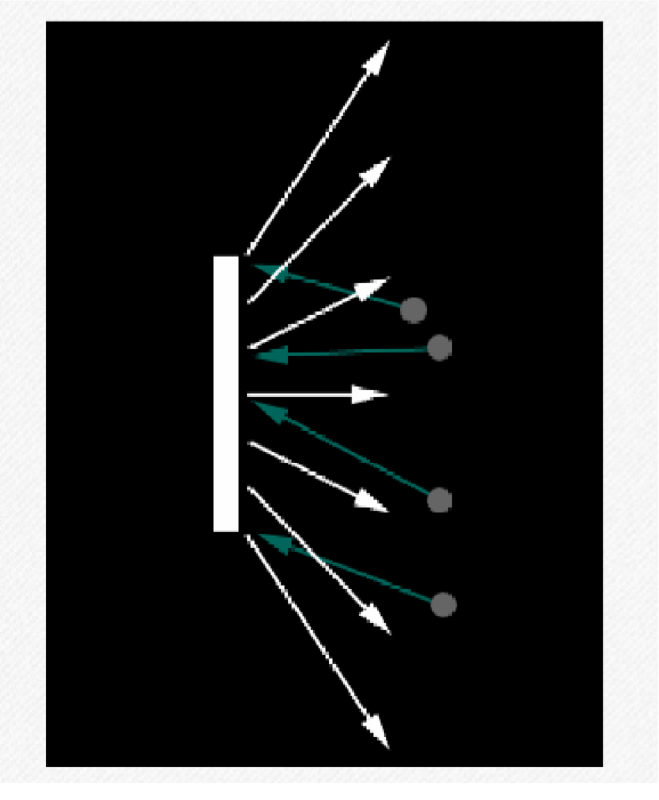
- Draw the right paddle
- Reset the ball when it reaches right edge
- Make the ball bounce on the right paddle

Scores

- Add one text on the upper left screen, one text on the upper right screen
- Register each "player" score, and display it in the texts

Bounce control

- Now, when the ball bounces on the paddle, we want its vertical speed to vary in function of the distance to center :



Opponent control

- Make the opponent paddle move up when the ball is above it, to move down when the ball is below it
- Set a maximum speed to the opponent paddle, so it can miss the ball
- In your code, this speed parameter should be easily accessible

Fix opponent shaking

- Create a deadzone on the paddle, so it moves only if the ball is outside this zone
- This will fix paddle shaking
- In your code, this deadzone length parameter should be easily accessible

Victory score

- Set a maximum score parameter to win the game
- Pause the game after victory to announce the winner
- Reset the game after an interaction (e.g.: mouse click). Indicate the needed interaction.

Polish

- Draw a dashed line in the middle of the screen

Advices from uncle Scott

- Advice 3: use *const* whenever possible

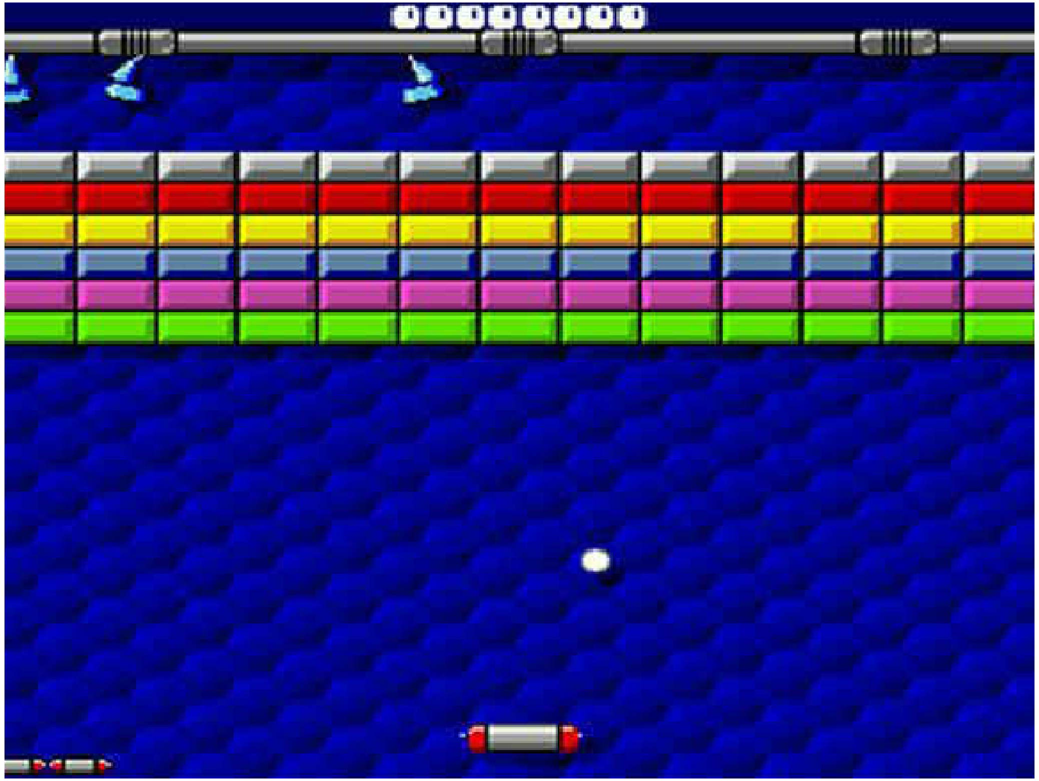
- Advice 4: initialise what you use and use member initialisation

```
Adress::Adress(street, number, city) :
          street("rue Gambetta"),
          number(2),
          city("Montpellier")
{
          // No assignment
}
```

- Advice 5: know that compilers silently generate default contructor, copy constructor, copy assignment operator and destructor

```
class Empty {
public:
          Empty() { ... }
          Empty(const Empty& obj) { ... }
          ~Empty() { ... }
          Empty& operator=(const Empty& obj)
{ ... }
}
```

Brick Breaker



A good start

- Create a black window with a ball bouncing on the edges inside

Paddle

- Reset the game if the ball cross the bottom edge

- Create a mouse-movable paddle that would make the ball bounce if it collides it

- Bouncing angle shall vary with the position of the ball on the paddle (the further on the paddle, the bigger the angle)

- Set the paddle vertical position 10% away from the bottom of the screen

Paddle fix

- When the ball hit the paddle from the side, it may have a strange behaviour

- Increase the paddle thickness to highlight this behaviour (only for tests)

- Fix it by making the paddle affect the ball only if the ball go downward

Another brick in the wall

- Create bricks on the top of the screen

- 14 rows, 10 cols of bricks, which dimensions are 80 (width) * 20 (height)

- Make a visual gap of 2 pixels on the right side and on the bottom of the bricks

- Storebricks in a unidimensional array (or std::vector)

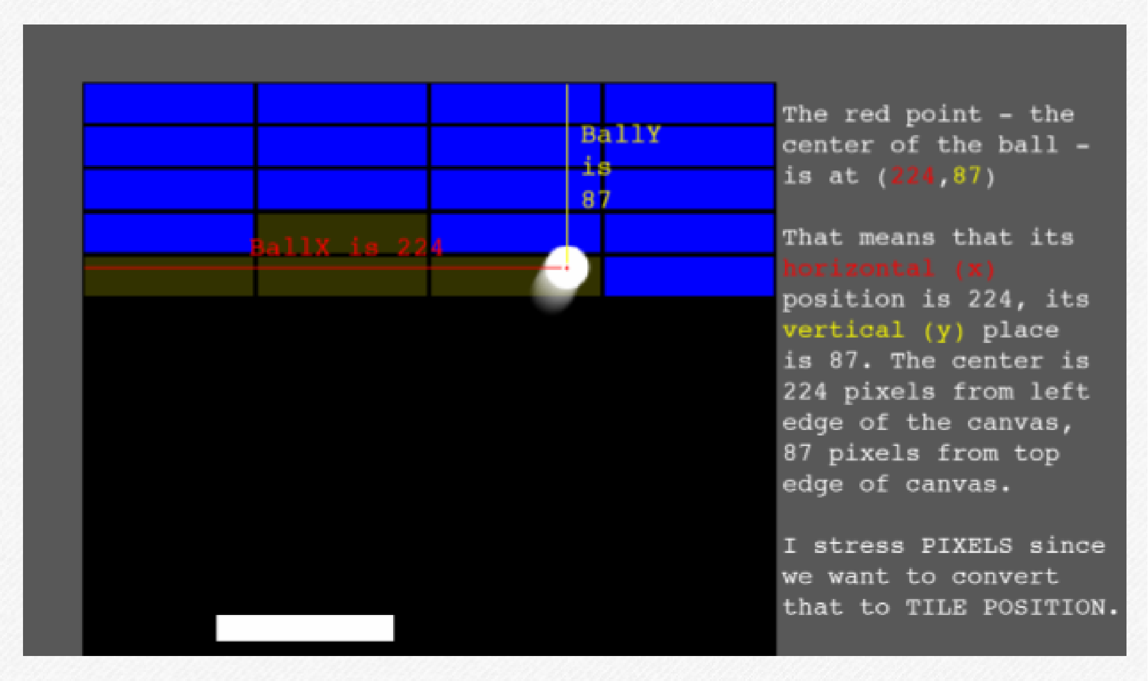
- Then you can use (brickCol + BRICK_COLS_NUMBER*brickRow) to convert from bidimensional to unidimensional coordinate

- Bonus: compare arrays and std::vector as datastructures in c++

Brick collision

- We could check if the ball were inside the brick, or not outside the brick. But it would be a lot of tests each frame.

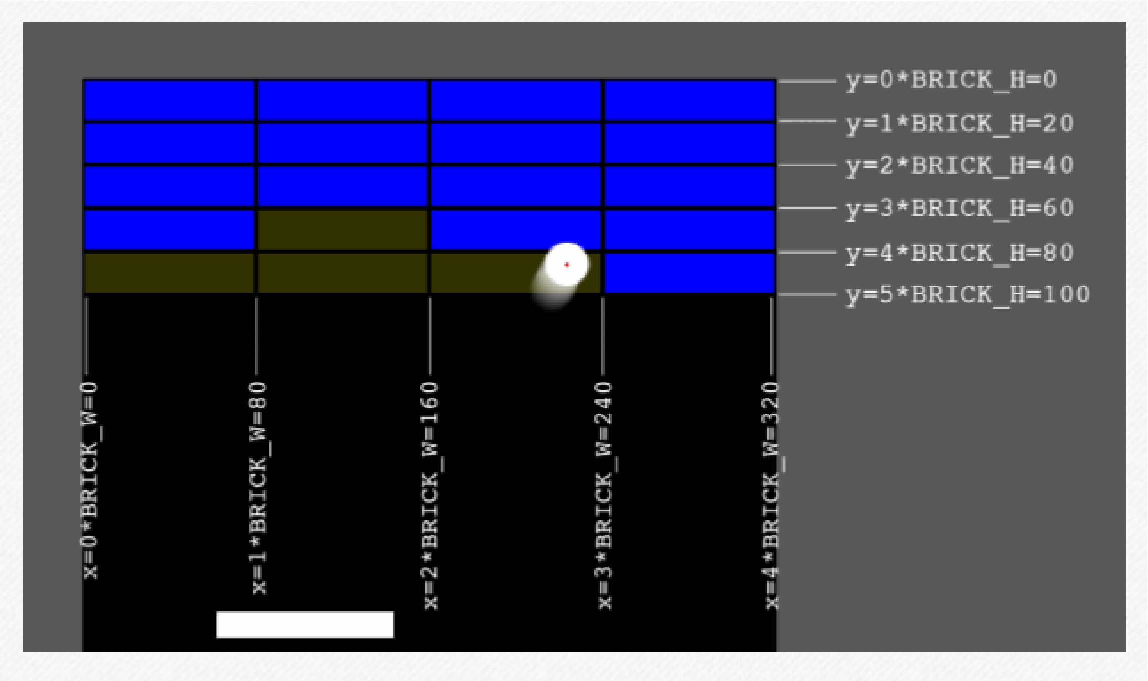
- We rather use a method to know where a point (the ball's center) is on a grid (the grid of bricks).



The red point - the center of the ball - is at (224,87)

That means that its horizontal (x) position is 224, its vertical (y) place is 87. The center is 224 pixels from left edge of the canvas, 87 pixels from top edge of canvas.

I stress PIXELS since we want to convert that to TILE POSITION.



- Dividing the ball coordinates by the bricks' width / height will give you its position on the grid

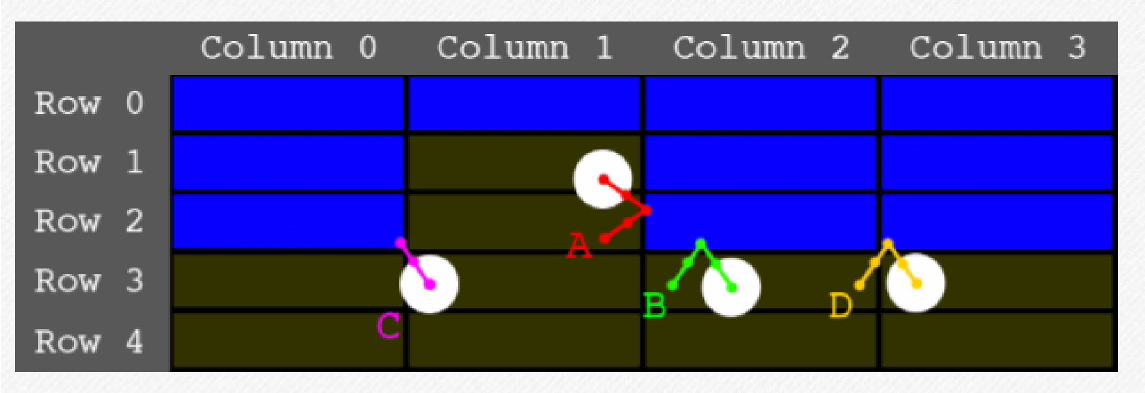
- Erase bricks that "are touched" by the ball, without making the ball bounce

- This method is REALLY important, keep it in mind for similar problems

Ball bounces on bricks

- Inverse the ball's vertical speed when a brick is hit

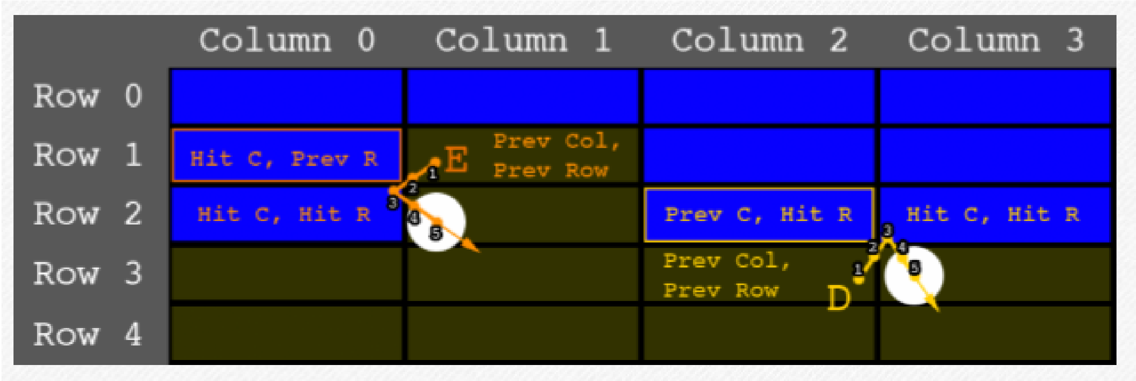
- But we would like different bounces directions if the ball hit an other side of a brick



- The key to solve the problem is to check on which part of the grid the ball was on the frame before (using the ball's speed)

- Did it change column (A) ? Did it change row (B) ? Both (C) ?

- D is a special case. There is also case E, D's vertical counterpart :



- We don't want the C scenario to happen in this case. The difference with C is we cross a diagonal (because of speed) before the collision

- We handle this case by logging the column or row of the previous frame

- There is a last case, where both adjacent sides are blocked by bricks. It only happens in corners, and we should reverse direction like with C

- Debug by adding a code to move the ball to the mouse position with left click, and change vertical/horizontal speed with right click

End game

- Reset bricks when the last one is gone

- Efficiency : keep a counter of the "living" brick number, instead of checking all the array each frame

- Remove the first three rows, and the debug code

Advices from uncle Scott

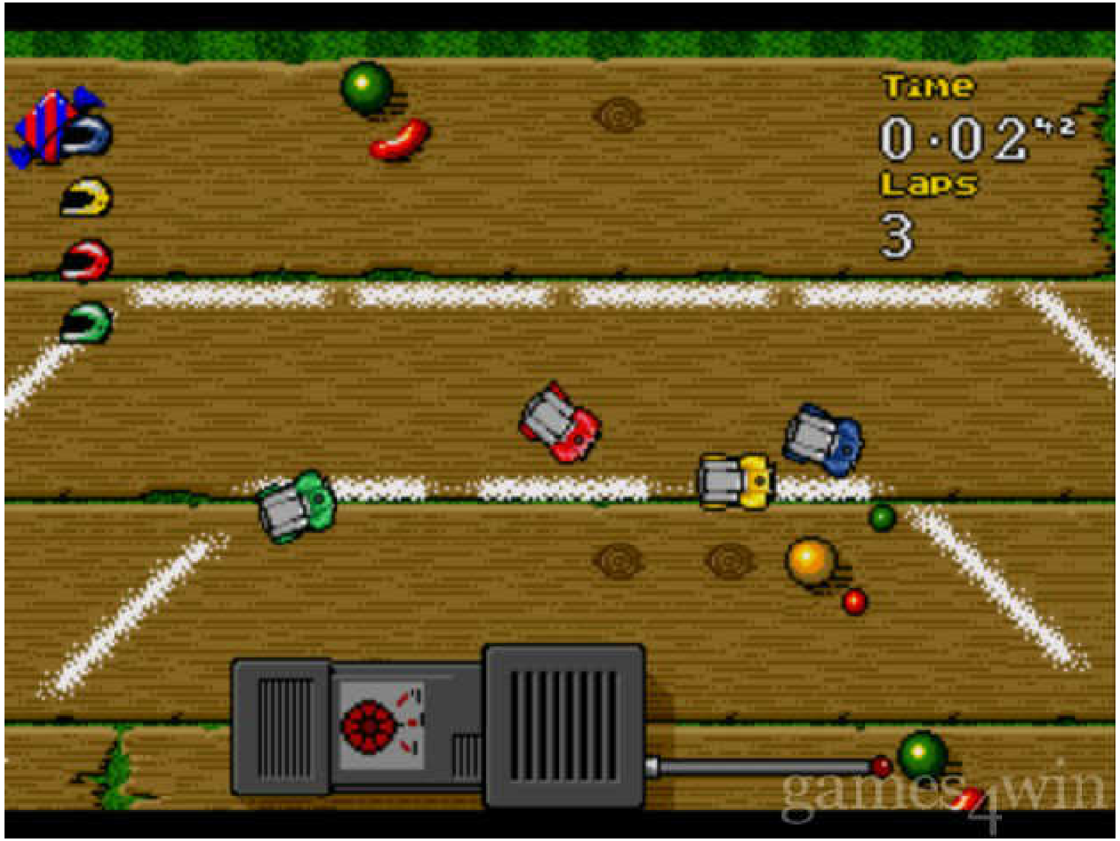
- Advice 6: explicitely disallow the use of computer generated function you don't want (declare them without body).

- Advice 7: declare destructor virtual in base classes (only when there is inheritance).

- Advice 8: prevent exception from leaving destructor. Either swallow or terminate the program.

- Advice 9 : never call virtual functions during construction or destruction.

Top Down Racer



Prepare project from previous code

- Copy the Brick Breaker project

- Delete all the paddle logic

- Resize the brick wall parameters to fill all the screen with bricks. Set the brick size to 40 * 40, so there will be 20 cols and 15 rows

- Rename everything that is "brick" to "tile", and "ball" to "car"

- Remove the brick removal mecanism

Tile grid

- We will create a tile grid from an array like this one :

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1,
 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1,
 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1,
 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1,
 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1,
 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1,
 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1,
 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1,
 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1,
 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1,
 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

- Load a tile when there is a 1, and no tile when there is zero

Car class

- You car class should load and draw an image to represent the car

- It should have an angle, in degrees, to set its rotation

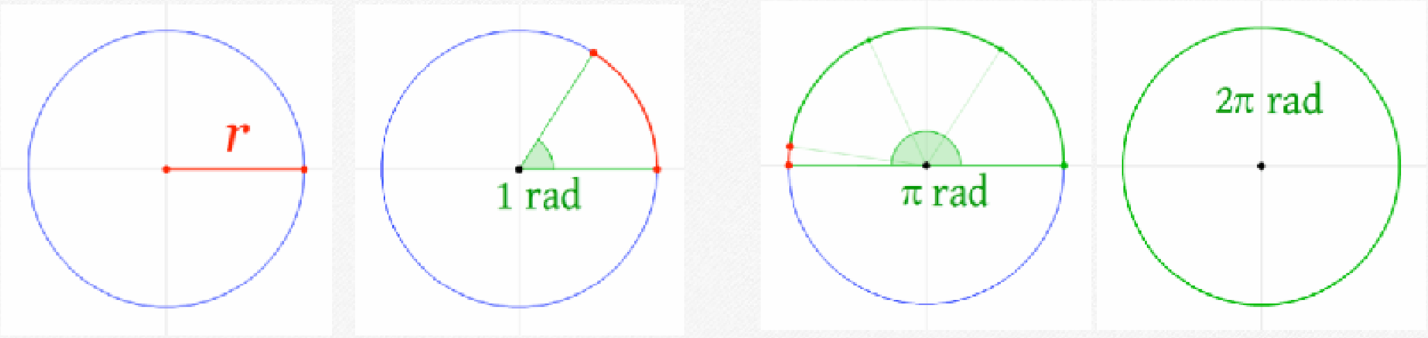
- Draw the car in the right direction with its rotation

- The car has a speed, that should increase when the UP key is pressed, and decrease when the DOWN key is pressed

- The car will moves thanks to this algorithm:

```
x += Math.cos(angle) * carSpeed;
y += Math.sin(angle) * carSpeed;
```

- Convert degrees to radian if needed :



Circle showing its radius, r *Radius laid along edge, 1 radian* *3.14159... (pi) radians is 180°* *So 2*pi is one full spin, or 360°*

Turning and friction

- When you hold the left key increase the angle by some degrees

- When you hold the right key decrease the angle by some degrees

- Add condition to make the car able to turn only if speed reaches a minimal absolute value

- Add a friction factor to make the car stop automatically. When speed reach a very small value, set it to zero :

```
speed = friction * speed; // With friction a value just under 1, like 0.95
```

Car start position

- Add a number 2 in the grid.

- Set the car to start at the position of this two in your game. You can use, with i the rank of the tile in the array :

```
int carRow = Math.floor(i/TILE_COLS);
int carCol = i % TILE_COLS
```

Car collisions

- Create a function that returns true if the position passed as argument is inside a tile which value is 1

- Use enums to set that the tile 0 is a Road, the tile 1 is a Block and the tile 2 is the PlayerStart

- Use the car NEXT position to test if next position is goind to be in a wall. To get next position :

```
int nextX = x + Math.cos(angle) * speed;
int nextY = Y + Math.sin(angle) * speed
```

- When the next position is colliding a block, multiply the car speed by -0.5 (inverse direction + speed decrease)

Image resource manager

- Create tiles images for road and for blocks

- Create a ResourceManager class that will load and contain all game images and identify them with a string key

- Now, when you want to draw the track tiles, use pointers to those images (or better: unique pointers)

- Change the car class so it uses this ResourceManager

An optimised way to draw tiles

- If you drew tiles columns by colums then line by line, do the contrary.

- We will use the grid array (which contains tiles line after line in a linear way) to draw our track

- Instead of computing each top left point of the tile to draw, we will use the natural order of the array to setup the tiles, in a Tile array :

```
int tile_y = 0;
int tile_x = 0;
int tile_index = 0;
for (i = 0; i < Track.TRACK_ROWS; i++) {
    tile_x = 0
    for (j = 0; j < Track.TRACK_COLS; j++) {
        int tile_type = TileId(Track.GRID[tile_index]);
        tiles[index] = new Tile(tile_x, tile_y, tile_type);
        // For next iteration
        tile_x += Track.TILE_WIDTH;
        ++tile_index;
    }
    tile_y += Track.TILE_HEIGHT;
}
```

- If you didn't until now, use a Tile class with the above constructor to draw your track !

Second player

- Add a second car for a second player, with different control keys
- When either of the players cross the finishing line, congratulate him or her
- Reset cars when a player hit a key after the congratulation