

An history of fundamental game algorithms with C++



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>

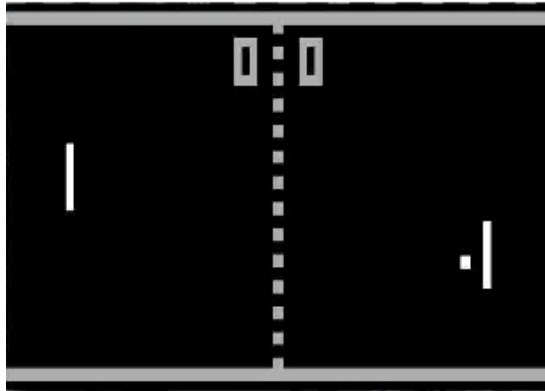
SDL2 image, mixer, ttf : <https://www.libsdl.org/projects/>

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

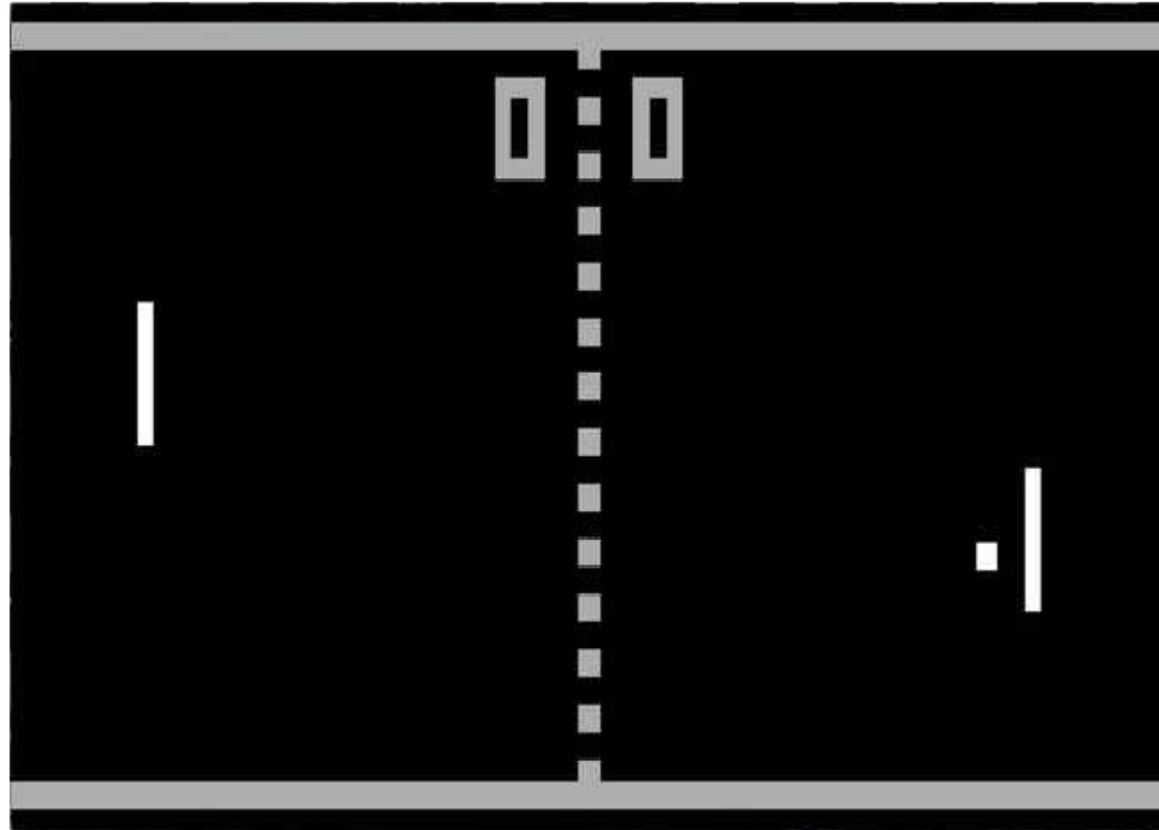


Two part for each project :

- 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

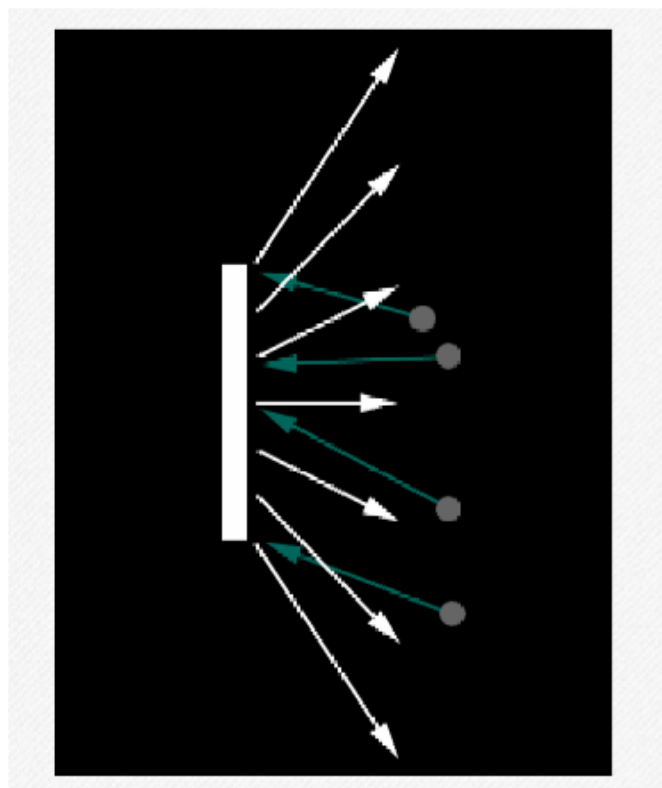
- Draw the right paddle
- Reset the ball when it reaches right edge
- Make the ball bounce on the right paddle

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 constructor, 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

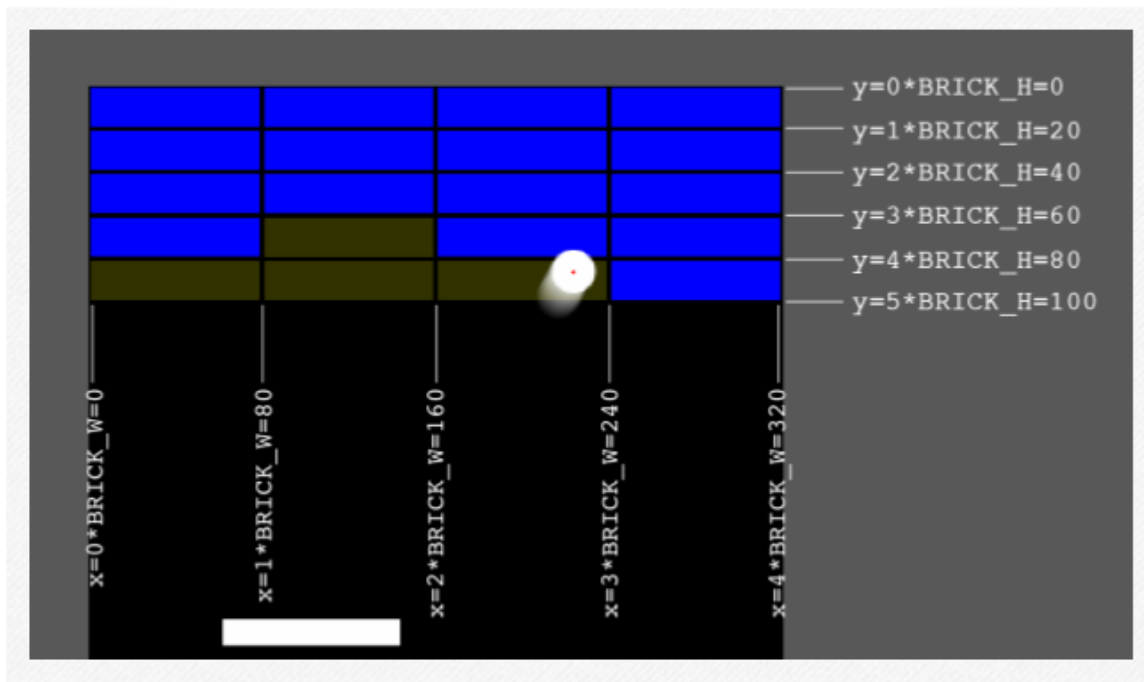
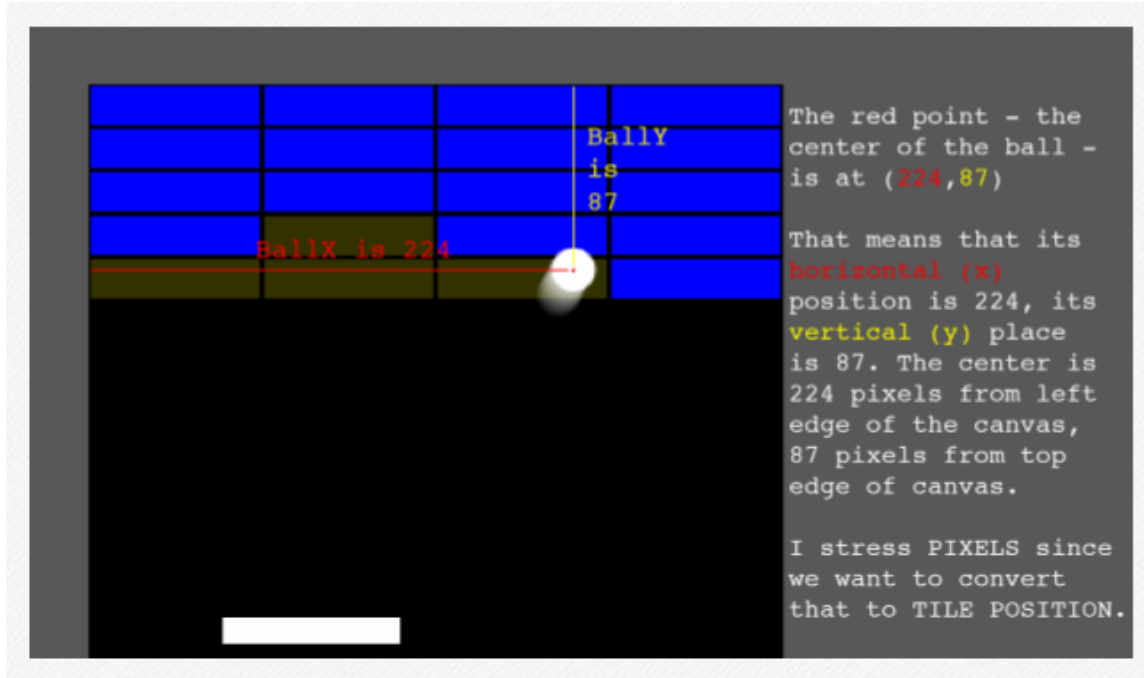
- When the ball hit the paddle from the side, it may have a strange behaviour
- Increase the paddle thickness to highlight this behaviour (only for tests)
- Fix it by making the paddle affect the ball only if the ball go downward

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
- Store bricks in a unidimensional array (or `std::vector`)
- Then you can use $(\text{brickCol} + \text{BRICK_COLS_NUMBER} * \text{brickRow})$ to convert from bidimensional to unidimensional coordinate
- Bonus: compare arrays and `std::vector` as datastructures in c++

Brick collision

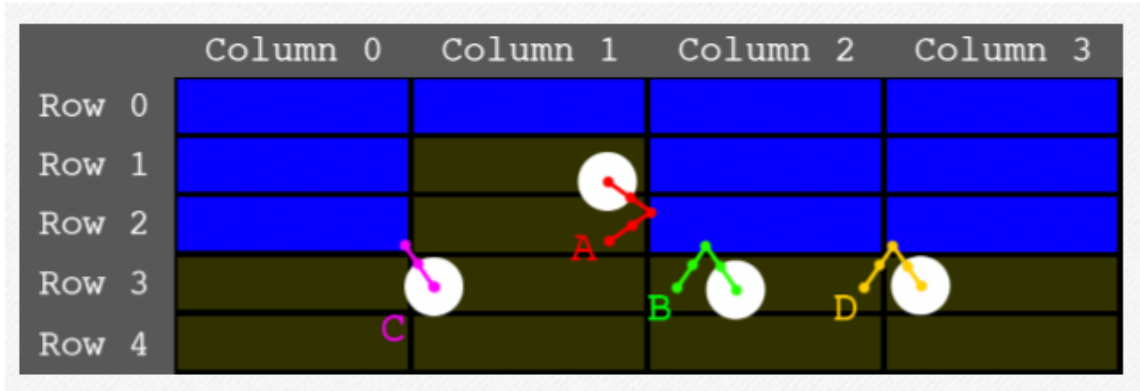
- We could check if the ball were inside the brick, or not outside the brick. But it would be a lot of tests each frame.
- We rather use a method to know where a point (the ball's center) is on a grid (the grid of bricks).



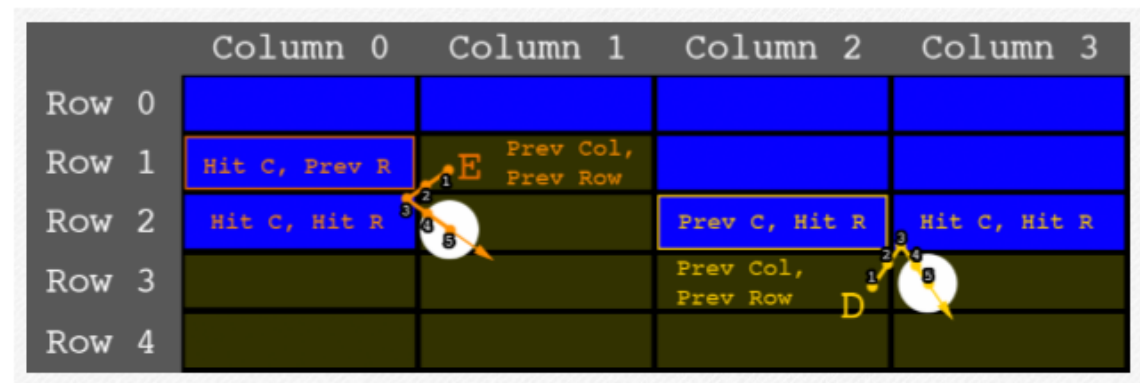
- Dividing the ball coordinates by the bricks' width / height will give you its position on the grid
- Erase bricks that "are touched" by the ball, without making the ball bounce
- This method is REALLY important, keep it in mind for similar problems

Ball bounces on bricks

- Inverse the ball's vertical speed when a brick is hit
- But we would like different bounce 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: explicitly 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.