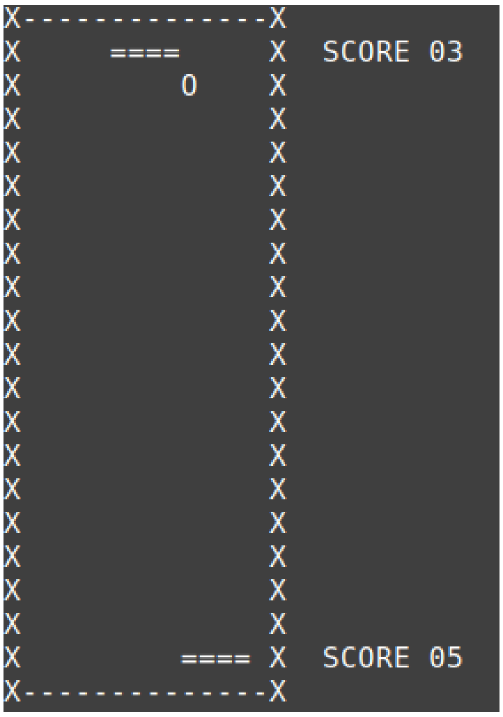
Programming Systems and Environments – Lab 10

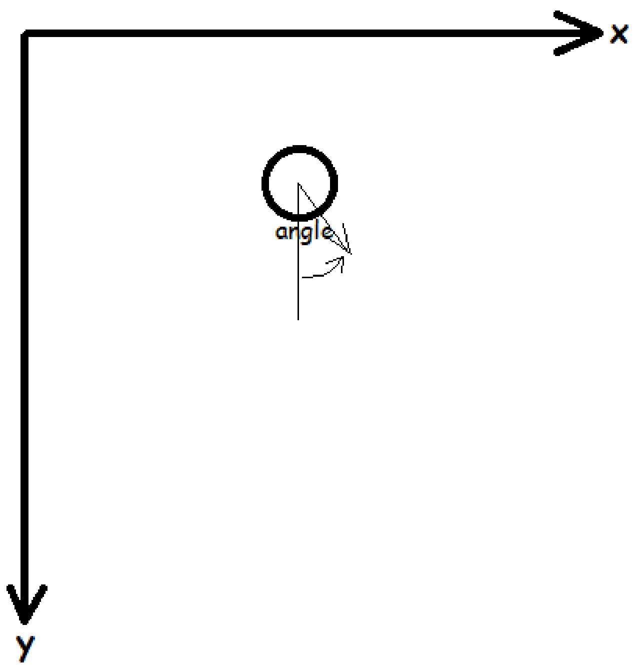
Jakub Grzana, 241530

Pong

That was challenging task. I've decided to recreate the game of pong with usage of floating point precision for position and angle of ball, defined by user size of arena (length and width) and score counter/display. Final program does work and allows two players to enjoy it (both paddles are human controlled) but code is terrible, and collision system is pretty broken,



```
X--------------X
X     ====     X  SCORE 03
X         0    X
X              X
X              X
X              X
X              X
X              X
X              X
X              X
X              X
X              X
X              X
X              X
X              X
X              X
X              X
X              X
X          ==== X  SCORE 05
X--------------X
```



I've created separate thread to display all Printable objects (used polymorphism – which I don't like cuz I had to use bare pointers, I don't know how to do this properly in C++), one to manage user input (movement of paddles), one for movement of ball, and one to "listen for" ball entering point lines.

I've programmed paddles as walls that can be moved around, that makes sense because this way I didn't have to make separate function for collision of ball with paddle. As sideeffect, I had to add mutex to RectangularWall class – movement would invalidate iterators within this object, causing segmentation fault during printing.

While visualization uses integer values of position, ball actually moves with floating point precision – it's just rounded mathematically for the sake of display.

Collision system

It's oversimplified, I couldn't find any proper solution on the internet and I didn't bother to overthink it. Basically, with every move of ball I calculate next position of ball, and I round it mathematically, like for visualization. Then I use the fact that RectangularObject is rectangle without any angle shift, so I can easily establish whether collision happens or not. Here's pseudocode

```
$const\ stepsize;$
$var\ pos_x, pos_y, angle;$
$var\ wall_{minx}, wall_{maxx}, wall_{miny}, wall_{maxy};$
$nextPos_x = round(pos_x + stepsize * \sin(angle));$
$nextPos_y = round(pos_y + stepsize * \cos(angle));$
$if \begin{cases} wall_{minx} \leq nextPos_x \leq wall_{maxx} \\ wall_{miny} \leq nextPos_y \leq wall_{maxy} \end{cases} then\ collision\ occured$
```

However detection of collision is one thing, reaction is another. For this I've used equally simple solution: I've made separate cases for ball approaching from leftside, rightside, behind or above.

This system seems to work properly with stationary walls, but it seems to break with paddles. Several times ball pierced through paddle, also I could've notice that ball reflects from paddle without touching it (being 1 point on grid ahead) which also isn't supposed to happen.

It could be explained by rounding, BUT for stepsize < 1 (and I'm using 0.4) this system should always work correctly.

Another possibility is that collision locks mutex of ball, but not of the walls, so for moving walls it can get wrong values. This would also explain why this error occurs only for paddles, not for walls. I've temporarily added locking mutex for wall in collision, but it didn't seem to change much, and it further obfuscated code.

So the final answer is: I don't know why this glitch happens.

Bibliography:

C++ reference: https://en.cppreference.com/w/
I didn't use anything more. All comes from my head.

CODE:

```
#include <ncurses.h>
#include <thread>
#include <list>
#include <chrono>
#include <iostream>
#include <random>
#include <cstdlib>
#include <cmath>
#include <mutex>

class Tools
{
	public: static int RandomInt(const int& min, const int& max) {
		static thread_local std::random_device rd;
		static thread_local std::mt19937 generator(rd());
		std::uniform_int_distribution<int> distribution(min, max);
		return distribution(generator);
	}

	public: static bool float_compare(const float& a, const float& b)
	{
		if(abs(a-b) <= 0.001) return true;
		return false;
	}

	public: static float deg2rad(const float& degs)
	{
		return degs * M_PI/180;
	}

	public: static float rad2deg(const float& rads)
	{
		return rads * 180/M_PI;
	}

	public: class Printable {
		public: virtual void Print() = 0;
	};

	public: class GraphicalObject : public Printable
	{
		private:
			int x;
			int y;
```

```
                        char letter;
                        bool downward;
                public:
                        int GetPosX() const { return this->x; }
                        int GetPosY() const { return this->y; }
                        char GetLetter() const { return this->letter; }
                        void SetPosX(int pos) { this->x = pos; }
                        void SetPosY(int pos) { this->y = pos; }
                        void SetLetter(char letter) { this->letter = letter; }
                        void Print() override { mvprintw(this->GetPosY(), this->GetPosX(),
"%c", this->GetLetter()); }
                        GraphicalObject() = delete;
                        GraphicalObject(int x, int y, char letter) { this->x = x; this->y = y;
this->letter = letter; }
        };

        public: Tools() = delete;
};

class RectangularWall : public Tools::Printable
{
        private:
                std::list<Tools::GraphicalObject> objs;
                char letter;
                int top_left_x, top_left_y, size_x, size_y;
                std::mutex lock;
                void Recreate()
                {
                        this->objs.clear();
                        for(int i = top_left_x; i <= top_left_x + size_x; ++i)
                        {
                                for(int j = top_left_y; j <= top_left_y + size_y; ++j)
                                {
                                        this->objs.push_back( Tools::GraphicalObject(i, j,
letter) );
                                }
                        }
                }
        public:
                RectangularWall() = delete;
                std::mutex& mutex() { return this->lock; }
                int TopLeftX() const { return top_left_x; }
                int TopLeftY() const { return top_left_y; }
                int TopRightX() const {       return top_left_x + size_x; }
                int TopRightY() const { return top_left_y; }
                int BottomLeftX() const { return this->top_left_x; }
                int BottomLeftY() const { return this->top_left_y + size_y; }
                int BottomRightX() const { return this->top_left_x + size_x; }
                int BottomRightY() const { return this->top_left_y + size_y; }
                int MinX() const { return this->top_left_x; }
                int MinY() const { return this->top_left_y; }
                int MaxX() const { return this->top_left_x + this->size_x; }
                int MaxY() const { return this->top_left_y + this->size_y; }
                void Print() override
                {
                        std::lock_guard<std::mutex> lock(this->mutex());
                        for(auto& obj : objs)
                        {
                                obj.Print();
                        }
                }
                void Move(int dx, int dy)
                {
                        std::lock_guard<std::mutex> lock(this->mutex());
                        this->top_left_x = this->top_left_x + dx;
                        this->top_left_y = this->top_left_y + dy;
                        this->Recreate();
                }
                RectangularWall(char letter, int top_left_x, int top_left_y, int size_x,
int size_y)
                {
                        this->top_left_x = top_left_x;
                        this->top_left_y = top_left_y;
                        this->size_x = size_x;
                        this->size_y = size_y;
                        this->letter = letter;
                        this->Recreate();
                }
};

class Ball : public Tools::GraphicalObject
{
        private:
                constexpr static float stepsize = 0.4; // the higher this parameter, the
higher chance for failing of collision check
                float precise_pos_x;
                float precise_pos_y;
                float precise_angle_deg;
                std::mutex lock;
                float dx() {
```

```
			return stepsize * sin(Tools::deg2rad(this->precise_angle_deg));
		}
		float dy() {
			return stepsize * cos(Tools::deg2rad(this->precise_angle_deg));
		}
		void SetPosX(int pos);
		void SetPosY(int pos);
	public:
		std::mutex& mutex() { return this->lock; }
		float GetPosX() const { return this->precise_pos_x; }
		float GetPosY() const { return this->precise_pos_y; }
		float GetAngle() const { return this->precise_angle_deg; }
		void ChangeAnglePrecise(float angle)
		{
			this->precise_angle_deg = angle;
		}
		void ChangeAngle(float angle)
		{
			this->precise_angle_deg = angle;
		}
		void Move()
		{
			this->precise_pos_x = this->GetPosX() + this->dx();
			this->precise_pos_y = this->GetPosY() + this->dy();
			GraphicalObject::SetPosX(round(this->precise_pos_x));
			GraphicalObject::SetPosY(round(this->precise_pos_y));
		}
		Ball() = delete;
		Ball(int x, int y, char letter) : GraphicalObject(x,y,letter) {
			SetPosition(x,y,60);
		}
		float Collision(const RectangularWall& wall) // return new angle
		{
			int expected_x = round(this->GetPosX() + this->dx());
			int expected_y = round(this->GetPosY() + this->dy());
			if(expected_x >= wall.MinX() && expected_x <= wall.MaxX() &&
expected_y >= wall.MinY() && expected_y <= wall.MaxY()) // collision occured
			{
				if(expected_x < GraphicalObject::GetPosX()) // approaching
from right to left
				{
					return 360 - this->GetAngle();
				}
				else if(expected_x > GraphicalObject::GetPosX()) //
approaching from left to right
```

```
                        {
                                return 360 - this->GetAngle();
                        }
                        else if(expected_y > GraphicalObject::GetPosY()) //
approaching from top
                        {
                                return 180 - this->GetAngle();
                        }
                        else // approaching from bottom
                        {
                                return 180 - this->GetAngle();
                        }
                }
                return this->GetAngle();
        }
        bool ApplyCollision(const RectangularWall& wall)
        {
                float new_angle = Collision(wall);
                if(Tools::float_compare(new_angle, this->GetAngle()))
                {
                        return false;
                }
                else
                {
                        this->ChangeAngle(new_angle);
                        return true;
                }
        }
        void SetPosition(int x, int y, float angle)
        {
                precise_pos_x = x;
                precise_pos_y = y;
                precise_angle_deg = angle;
        }
};

void print(const bool& stop, std::list<Tools::Printable*>& objs)
{
        curs_set(0);
        while(!stop)
        {
                erase();
                for(const auto& obj : objs) { obj->Print(); }
                refresh();
        }
```

```
}

void ball_movement(const bool& stop, Ball& ball, const std::list<RectangularWall*>&
walls, const float& speed)
{
	constexpr float tick_delay = 80;
	int tick = tick_delay / speed;
	while(!stop)
	{
		std::this_thread::sleep_for(std::chrono::milliseconds(tick));
		std::lock_guard<std::mutex> lock(ball.mutex());
		for(const auto& wall : walls)
		{
			ball.ApplyCollision(*wall);
		}
		ball.Move();
	}
}

void player_input(bool& stop, RectangularWall& top_paddle, RectangularWall&
bottom_paddle, const int map_width)
{
	constexpr int tick = 50;
	while(!stop)
	{
		//std::this_thread::sleep_for(std::chrono::milliseconds(tick));
		char letter = getchar();
		switch(letter)
		{
			case 'a': {
				if(top_paddle.MinX() > 1) top_paddle.Move(-1, 0);
				} break;
			case 'd': {
				if(top_paddle.MaxX() < map_width - 1) top_paddle.Move(1,
0);
				} break;
			case 'j': {
				if(bottom_paddle.MinX() > 1) bottom_paddle.Move(-1, 0);
				} break;
			case 'l': {
				if(bottom_paddle.MaxX() < map_width - 1)
bottom_paddle.Move(1, 0);
				} break;
			case 'G': {
				stop = true;
```

```
                        } break;
                }
        }
}

class ScoreDisplay : public Tools::Printable
{
        private:
                std::list<Tools::GraphicalObject> objs;
                std::mutex lock;
                int top_left_x, top_left_y;
                void Recreate(int score)
                {
                        std::lock_guard<std::mutex> lock(this->mutex());
                        this->objs.clear();
                        this->objs.push_back( Tools::GraphicalObject( top_left_x,
top_left_y, 'S') );
                        this->objs.push_back( Tools::GraphicalObject( top_left_x+1,
top_left_y, 'C') );
                        this->objs.push_back( Tools::GraphicalObject( top_left_x+2,
top_left_y, 'O') );
                        this->objs.push_back( Tools::GraphicalObject( top_left_x+3,
top_left_y, 'R') );
                        this->objs.push_back( Tools::GraphicalObject( top_left_x+4,
top_left_y, 'E') );
                        this->objs.push_back( Tools::GraphicalObject( top_left_x+6,
top_left_y, 48 + (score%100)/10) );
                        this->objs.push_back( Tools::GraphicalObject( top_left_x+7,
top_left_y, 48 + score%10) );

                }
        public:
                std::mutex& mutex() { return this->lock; }
                void Print() override
                {
                        std::lock_guard<std::mutex> lock(this->mutex());
                        for(auto& obj : objs)
                        {
                                obj.Print();
                        }
                }
                ScoreDisplay(int x, int y)
                {
                        this->top_left_x = x;
                        this->top_left_y = y;
                        this->Recreate(0);
                }
                void Set(int score)
                {
                        this->Recreate(score);
                }
};

void point_listener(const bool& stop, Ball& ball,
        const RectangularWall& top_line, const RectangularWall& bottom_line,
        int& score_top, int& score_bottom,
        int& map_width, int& map_length,
        ScoreDisplay& top, ScoreDisplay& bottom)

{
        constexpr int tick = 10;
        while(!stop)
        {
                std::this_thread::sleep_for(std::chrono::milliseconds(tick));
                std::lock_guard<std::mutex> lock(ball.mutex());
                if(ball.Collision(top_line) != ball.GetAngle())
                {
                        score_bottom++;
                        bottom.Set(score_bottom);
                        ball.ApplyCollision(top_line);
                        ball.SetPosition(map_width/2, map_length/2, ball.GetAngle());
                }
                else if(ball.Collision(bottom_line) != ball.GetAngle())
                {
                        score_top++;
                        top.Set(score_top);
                        ball.ApplyCollision(bottom_line);
                        ball.SetPosition(map_width/2, map_length/2, ball.GetAngle());
                }
        }
}


int PingPong(int map_width, int map_length, int paddle_size, float speed)
{
        initscr();

        bool stop = false;
        int score_top = 0;
        int score_bottom = 0;
```

```
	std::list<std::thread> threads;
	std::list<Tools::Printable*> all_objs;
	std::list<RectangularWall*> walls;
	Ball ball(map_width / 2, map_length / 2, 'O');

	RectangularWall top_player_line('-', 0, 0, map_width, 0) ;
all_objs.push_back(&top_player_line);
	RectangularWall bottom_player_line('-', 0, map_length, map_width, 0);
all_objs.push_back(&bottom_player_line);
	ScoreDisplay top(map_width+3, 1); all_objs.push_back(&top);
	ScoreDisplay bottom(map_width+3, map_length-1); all_objs.push_back(&bottom);

	walls.push_back( new RectangularWall('X', 0, 0, 0, map_length) );
	walls.push_back( new RectangularWall('X', map_width, 0, 0, map_length) );
	walls.push_back( new RectangularWall('=', map_width/2 - paddle_size/2, 1,
paddle_size-1, 0) ); RectangularWall& top_paddle = *(walls.back());
	walls.push_back( new RectangularWall('=', map_width/2 - paddle_size/2,
map_length-1, paddle_size-1, 0) ); RectangularWall& bottom_paddle = *(walls.back());

	for(auto& wall : walls)
	{
		all_objs.push_back(wall);
	}
	all_objs.push_back(&ball);

	threads.push_back( std::thread( player_input, std::ref(stop),
std::ref(top_paddle), std::ref(bottom_paddle), std::ref(map_width) ));
	threads.push_back( std::thread( print, std::ref(stop), std::ref(all_objs) ));
	threads.push_back( std::thread( ball_movement, std::ref(stop), std::ref(ball),
std::ref(walls), std::ref(speed) ));
	threads.push_back( std::thread( point_listener, std::ref(stop), std::ref(ball),
std::ref(top_player_line), std::ref(bottom_player_line), std::ref(score_top),
std::ref(score_bottom), std::ref(map_width), std::ref(map_length), std::ref(top),
std::ref(bottom) ));

	for(auto& thread : threads)
	{ thread.join(); }

	endwin();
	return 0;
}

int main()
{
        std::cout << "a/d to move top platform" << std::endl
        << "l/m to move bottom plaftorm" << std::endl
        << "G to quit" << std::endl
        << std::endl
        << "Type in map length, width and speed modifier" << std::endl;
        int map_length = 0, map_width = 0; std::cin >> map_length >> map_width;
        float speed = 1; std::cin >> speed;

        return PingPong(map_width, map_length, 4, speed);
}
```