ДОДАТОК А

Вихідний код програми

1) Swarm.h

```
/ *
* Copyright (c) 2021
* /

#ifndef SWARM_H
#define SWARM_H

#include <time.h>
#include <math.h>
#include "Boid.h"
#include "BoidManager.h"
#include "EnergyManager.h"
#include "ObsManager.h"
#include "SpikeManager.h"
#include "Vec.h"


class Swarm: public Test
{
public:

      Swarm ()
      {
            srand(Time (0));

            m_enableMouse = false;
            m_mousePos.SetZero ();

            CreateGround ();

            m_energyManager = new EnergyManager ();
            m_energyManager-> CreateEnergies (m_world,
m_groundWidth);

            m_obsManager = new ObsManager ();
            m_obsManager-> CreateEnergies (m_world, m_groundWidth);

            m_boidManager = new BoidManager (true, b2Color (0, 1, 0),
m_energyManager, m_obsManager);
            m_boidManager-> CreateBoids (m_world, "boid",
m_groundWidth);

            m_boidManagerSimple = new BoidManager (false, b2Color (1,
0, 0), m_energyManager, m_obsManager);
```

```
            m_boidManagerSimple-> CreateBoids (m_world, "simpleBoid",
m_groundWidth);

            m_spikeManager = new SpikeManager ();
      }

      ~ Swarm ()
      {
            delete m_boidManager;
            delete m_boidManagerSimple;
            delete m_energyManager;
            delete m_spikeManager;
      }

      void CreateGround ()
      {
            b2BodyDef bd;
            b2Body * ground = m_world-> CreateBody (& bd);
            ground-> SetUserData ((void *) "ground");

            b2EdgeShape shape;
            shape.Set (b2Vec2 (-m_groundWidth, -m_groundHeight),
b2Vec2 (m_groundWidth, -m_groundHeight)); // bottom
            ground-> CreateFixture (& shape, 0);

            shape.Set (b2Vec2 (m_groundWidth, -m_groundHeight),
b2Vec2 (m_groundWidth, m_groundHeight)); // right
            ground-> CreateFixture (& shape, 0);

            shape.Set (b2Vec2 (m_groundWidth, m_groundHeight), b2Vec2
(-m_groundWidth, m_groundHeight)); // top
            ground-> CreateFixture (& shape, 0);

            shape.Set (b2Vec2 (-m_groundWidth, m_groundHeight),
b2Vec2 (-m_groundWidth, -m_groundHeight)); // left
            ground-> CreateFixture (& shape, 0);
      }

      void MouseDown (const b2Vec2 & p)
      {
            m_enableMouse = true;

            m_boidManager-> UpdateMouseTarget (m_mousePos);
            m_boidManagerSimple-> UpdateMouseTarget (m_mousePos);
            m_energyManager-> UpdateMouseTarget (m_mousePos);
      }

      void MouseUp (const b2Vec2 & p)
      {
            m_enableMouse = false;
      }
```

```
	void MouseMove (const b2Vec2 & p)
	{
		m_mousePos = p;
	}

	void Keyboard (int key)
	{
		switch (key)
		{
		case GLFW_KEY_E:
			break;
		}
	}

	void BeginContact (b2Contact * contact)
	{
		if (contact-> IsTouching ())
		{
			b2Body * spikeBody = contact-> GetFixtureA () ->
GetBody ();
			b2Body * boidBody = contact-> GetFixtureB () ->
GetBody ();
			const char * spikeData = (const char *) spikeBody->
GetUserData ();
			const char * boidData = (const char *) boidBody->
GetUserData ();

			if (strcmp(SpikeData, "spike") == 0 && (strcmp
(boidData, "boid") == 0 || strcmp (boidData, "simpleBoid") == 0))
			{
				Boid * boid = m_boidManager-> FindBoidByBody
(boidBody);
				Boid * boidSimple = m_boidManagerSimple->
FindBoidByBody (boidBody);

				BoidSpikeContact (boid, spikeBody);
				BoidSpikeContact (boidSimple, spikeBody);
			}
		}
	}

	void BoidSpikeContact (Boid * boid, b2Body * spikeBody)
	{
		if (boid! = 0)
		{
			boid-> m_energy – = 10;

			m_spikeManager-> FindSpikeByBody (spikeBody) ->
SetHit (true);
		}
```

```
	}

	void Step (Settings * settings)
	{
		static int time = 0;
		time% = 100;

		if (time == 0)
		{
			m_spikeManager-> AddSpike (m_world, m_boidManager->
GetBoids (), m_boidManagerSimple-> GetBoids ());
			m_spikeManager-> AddSpike (m_world,
m_boidManagerSimple-> GetBoids (), m_boidManager-> GetBoids ());
		}

		m_boidManager-> UpdateBoids (m_mousePos, m_enableMouse);
		m_boidManagerSimple-> UpdateBoids (m_mousePos,
m_enableMouse);
		m_spikeManager-> UpdateSpikes ();

		m_boidManager-> Draw (m_mousePos);
		m_boidManagerSimple-> Draw (m_mousePos);
		m_energyManager-> Draw (m_mousePos);

		DrawStatistic ();

		time ++;

		Test :: Step (settings);
	}

	void DrawStatistic ()
	{
		double boidEnergy = 0, boidSimpleEnergy = 0;

		for (int i = 0; i <m_boidManager-> GetBoids ().size(); i
++)
		{
			boidEnergy + = m_boidManager-> GetBoids () [i] ->
m_energy;
		}

		for (int i = 0; i <m_boidManagerSimple-> GetBoids
().size(); i ++)
		{
			boidSimpleEnergy + = m_boidManagerSimple-> GetBoids
() [i] -> m_energy;
		}

		g_debugDraw.DrawString (5, m_textLine, "swarm boids
energy =% g", boidEnergy);
```

```
            m_textLine + = DRAW_STRING_NEW_LINE;
            g_debugDraw.DrawString (5, m_textLine, "simple boids
energy =% g", boidSimpleEnergy);
            m_textLine + = DRAW_STRING_NEW_LINE;
      }

      static Test * Create ()
      {
            return new Swarm;
      }

protected:

      const int m_groundWidth = 500;
      const int m_groundHeight = 500;

      bool m_enableMouse;
      b2Vec2 m_mousePos;

      BoidManager * m_boidManager;
      BoidManager * m_boidManagerSimple;
      EnergyManager * m_energyManager;
      ObsManager * m_obsManager;
      SpikeManager * m_spikeManager;
};

#endif
```

2) Vec.h

```
#ifndef VEC_H
#define VEC_H

#include <Box2D / Box2D.h>

class Vec
{
public:

      static double Distance (b2Vec2 A, b2Vec2 B)
      {
            return sqrt((Bx – Ax) * (Bx – Ax) + (By – Ay) * (By –
Ay));
      }

      static b2Vec2 Move (b2Vec2 A, b2Vec2 B, double dist)
      {
            return A + (B – A) * (dist / Distance (A, B));
      }
};
```

```
#endif

3) SimpleBody.h

#ifndef SIMPLE_BODY_H
#define SIMPLE_BODY_H

#include <Box2D / Box2D.h>
#include <vector>
#include "Vec.h"

using namespace std;

class SimpleBody
{
public:

      SimpleBody ();
      SimpleBody (b2World * world, bool dynamic, b2Vec2 pos, float
size);

      virtual void Draw (b2Vec2 mousePos);

      b2Body * GetBody () {return m_body; }
      b2Fixture * GetFixture () {return m_fixture; }

      double m_energy;

protected:

      b2Body * m_body;
      b2Fixture * m_fixture;
};

#endif

#include "SimpleBody.h"
#include "../../Framework/Test.h" // wtf omg

SimpleBody :: SimpleBody ()
{

}

SimpleBody :: SimpleBody (b2World * world, bool dynamic, b2Vec2 pos,
float size)
{
      m_energy = size * 10;

      b2BodyDef bd;
      bd.type = dynamic? b2_dynamicBody: b2_staticBody;
```

```
	bd.position.Set (pos.x, pos.y);
	m_body = world-> CreateBody (& bd);

	m_body-> SetLinearDamping (0.5f);
	m_body-> SetAngularDamping (0.5f);

	b2CircleShape shape;
	shape.m_radius = size;

	b2FixtureDef fd;
	fd.shape = & shape;
	fd.density = 1.f;
	fd.restitution = 0.2f;
	fd.friction = 1.f;
	m_fixture = m_body-> CreateFixture (& fd);
}

void SimpleBody :: Draw (b2Vec2 mousePos)
{
	b2Vec2 pos = g_camera.ConvertWorldToScreen (m_body->
GetWorldCenter ());
	g_debugDraw.DrawString (pos.x, pos.y, "% i", (int) m_energy);
}
```

4) Boid.h

```
#ifndef BOID_H
#define BOID_H

#include "SimpleBody.h"
#include "Energy.h"
#include "BoidManager.h"

class BoidManager;

class Boid: public SimpleBody
{
public:

	Boid (b2World * world, BoidManager * parent, const char *
userData, bool dynamic, b2Vec2 pos, float size, vector<Energy *>
energies);

	void UpdateTarget (vector<Energy *> energies);
	b2Vec2 Update (b2Vec2 mousePos, vector<Energy *> energies);
	b2Vec2 MoveToMouse (b2Vec2 mousePos);
	b2Vec2 MoveToEnergy ();

	void UpdateEnergy ();
	void SwapEnergy ();

	void Draw (b2Vec2 mousePos);
```

```
	void SetTarget (Energy * energyTarget) {m_energyTarget =
energyTarget; }
	Energy * GetTarget () {return m_energyTarget; }
	BoidManager * GetParent () {return m_parent; }

	int m_speed;
	int m_energyLimit;
	double m_fieldView;
	double m_energySwapSpeed;
	double m_energyLossSpeed;

	const int m_limit = 100; // some kind of a constant

protected:

	Energy * m_energyTarget;
	BoidManager * m_parent;
};

#endif

#include "Boid.h"
#include "../../Framework/Test.h" // wtf omg

Boid :: Boid (b2World * world, BoidManager * parent, const char *
userData, bool dynamic, b2Vec2 pos, float size, vector <Energy *>
energies): SimpleBody (world, dynamic, pos, size)
{
	m_parent = parent;
	m_body-> SetUserData ((void *) userData);

	m_energyTarget = 0;
	m_speed = 20;
	m_energyLimit = m_limit;
	m_fieldView = m_limit;
	m_energySwapSpeed = 0.01;
	m_energyLossSpeed = 0.001;

	UpdateTarget (energies);
}

void Boid :: UpdateTarget (vector <Energy *> energies)
{
	double minDist = DBL_MAX;
	b2Vec2 pos = m_body-> GetWorldCenter ();
	m_energyTarget = 0;

	for (int i = 0; i <energies.size (); i ++)
	{
```

```
            b2Vec2 energyPos = energies [i] -> GetBody () ->
GetWorldCenter ();
            double dist = Vec :: Distance (pos, energyPos);

            if (minDist> dist && energies [i] -> m_energy> 0 / * &&
dist <m_fieldView * /)
            {
                  minDist = dist;
                  m_energyTarget = energies [i];
            }
      }
}

b2Vec2 Boid :: Update (b2Vec2 mousePos, vector <Energy *> energies)
{
      if (m_body-> GetType () == b2_dynamicBody)
      {
            UpdateEnergy ();

            if (m_energyTarget! = 0)
            {
                  if (m_energyTarget-> m_energy> 0)
                  {
                        SwapEnergy ();

                        return MoveToEnergy ();
                  }
                  else
                  {
                        UpdateTarget (energies);
                  }
            }
      }

      return b2Vec2 (0, 0);
}

b2Vec2 Boid :: MoveToMouse (b2Vec2 mousePos)
{
      b2Vec2 vec = mousePos – m_body-> GetWorldCenter ();
      vec.Normalize ();

      return vec * m_speed;
}

b2Vec2 Boid :: MoveToEnergy ()
{
      b2Vec2 vec = m_energyTarget-> GetBody () -> GetWorldCenter () –
m_body-> GetWorldCenter ();
      vec.Normalize ();
```

```
      return vec * m_speed;
}

void Boid :: UpdateEnergy ()
{
      m_energy – = m_energyLossSpeed;

      if (m_energy <= 0)
      {
            m_body-> SetType (b2_staticBody);
      }
}

void Boid :: SwapEnergy ()
{
      b2Vec2 pos = m_body-> GetWorldCenter ();
      b2Vec2 targetPos = m_energyTarget-> GetBody () ->
GetWorldCenter ();
      float targetSize = m_energyTarget-> GetFixture () -> GetShape
() -> m_radius;

      if (Vec :: Distance (pos, targetPos) <targetSize * 5 &&
m_energyTarget-> m_energy> 0)
      {
            m_energy + = m_energySwapSpeed;
            m_energyTarget-> m_energy – = m_energySwapSpeed;
      }
}

void Boid :: Draw (b2Vec2 mousePos)
{
      SimpleBody :: Draw (mousePos);

      g_debugDraw.DrawCircle (m_body-> GetWorldCenter (),
m_fieldView, b2Color (0.4, 0.4, 0.4));
}
```

5) BoidManager.h

```
#ifndef BOID_MANAGER_H
#define BOID_MANAGER_H

#include "Boid.h"
#include "EnergyManager.h"
#include "ObsManager.h"

class Boid;

class BoidManager
{
public:
```

```
	BoidManager (bool isSwarm, b2Color color, EnergyManager *
energy, ObsManager * obs);

	void CreateBoids (b2World * world, const char * userData, int
groundSize);

	Energy * GetCommonTarget ();
	void UpdateMouseTarget (b2Vec2 mousePos);
	void UpdateBoids (b2Vec2 mousePos, bool enableMouse);
	void MoveBoids (b2Vec2 mousePos, bool enableMouse);
	void LimitSpeed (int j);
	b2Vec2 MoveToCenter (int j);
	b2Vec2 MoveFromNeighbors (int j);
	b2Vec2 MoveFromObstacle (int j);

	void Draw (b2Vec2 mousePos);

	Boid * FindBoidByBody (b2Body * body);
	vector<Boid *> GetBoids () {return m_boids; }
	bool IsSwarm () {return m_isSwarm; }

private:

	const int m_boidsCount = 10;

	vector<Boid *> m_boids;
	vector<Energy *> m_energies;
	vector<Obstacle *> m_obstacles;

	SimpleBody * m_mouseTarget;

	bool m_isSwarm;
	b2Color m_color;
};

#endif

#include "BoidManager.h"
#include "../../Framework/Test.h" // wtf omg

BoidManager :: BoidManager (bool isSwarm, b2Color color,
EnergyManager * energy, ObsManager * obs)
{
	m_energies = energy-> GetEnergies ();
	m_obstacles = obs-> GetObstacles ();
	m_mouseTarget = 0;
	m_isSwarm = isSwarm;
	m_color = color;
}
```

```
void BoidManager :: CreateBoids (b2World * world, const char *
userData, int groundSize)
{
	int x = -groundSize / 2 + rand()% (GroundSize);
	int y = -groundSize / 2 + rand()% (GroundSize);

	for (int i = 0; i <m_boidsCount; i ++)
	{
		int r = m_boidsCount * 10;

		b2Vec2 pos (x + -r + rand()% R * 2, y + -r + rand ()% r *
2);
		Boid * boid = new Boid (world, this, userData, true, pos,
1.f, M_energies);

		m_boids.push_back(Boid);
	}
}

Energy * BoidManager :: GetCommonTarget ()
{
	for (int i = 0; i <m_boids.size (); i ++)
	{
		if (m_boids [i] -> GetBody () -> GetType () ==
b2_dynamicBody)
		{
			if (m_boids [i] -> GetTarget ()! = 0 && m_boids [i]
-> GetTarget () -> m_energy> 0)
			{
				return m_boids [i] -> GetTarget ();
			}
		}
	}

	return 0;
}

void BoidManager :: UpdateMouseTarget (b2Vec2 mousePos)
{
	m_mouseTarget = 0;

	for (int i = 0; i <m_boids.size (); i ++)
	{
		if (m_boids [i] -> GetFixture () -> TestPoint (mousePos))
		{
			m_mouseTarget = m_boids [i];
		}
	}
}

void BoidManager :: UpdateBoids (b2Vec2 mousePos, bool enableMouse)
```

```
{
	MoveBoids (mousePos, enableMouse);
}

void BoidManager :: MoveBoids (b2Vec2 mousePos, bool enableMouse)
{

	for (int i = 0; i <m_boids.size (); i ++)
	{
		if (m_boids [i] -> GetBody () -> GetType () ==
b2_dynamicBody)
		{
			if (! enableMouse)
			{
				b2Vec2 energyVec = m_boids [i] -> Update
(mousePos, m_energies);

				if (m_isSwarm)
				{
					Energy * energy = GetCommonTarget ();
					m_boids [i] -> SetTarget (energy);
				}

				b2Vec2 obsVec = MoveFromObstacle (i);
				b2Vec2 centerVec = MoveToCenter (i);
				b2Vec2 neighVec = MoveFromNeighbors (i);

				b2Vec2 commonVec = energyVec + obsVec;
				commonVec.Normalize ();
				commonVec * = m_boids [i] -> m_speed;
				commonVec + = m_isSwarm? centerVec + neighVec:
b2Vec2 (0, 0);
				m_boids [i] -> GetBody () ->
ApplyForceToCenter (commonVec, true);
			}
			else
			{
				b2Vec2 mouseVec = m_boids [i] -> MoveToMouse
(mousePos);
				m_boids [i] -> GetBody () ->
ApplyForceToCenter (mouseVec, true);
			}

			LimitSpeed (i);
		}
	}
}

b2Vec2 BoidManager :: MoveToCenter (int j)
{
	b2Vec2 massCenter (0, 0);
```

```
	int boidsNum = 0;

	for (int i = 0; i <m_boids.size (); ++ i)
	{
		if (i! = j && m_boids [i] -> GetBody () -> GetType () ==
b2_dynamicBody)
		{
			massCenter + = m_boids [i] -> GetBody () ->
GetWorldCenter ();
			++ boidsNum;
		}
	}

	if (boidsNum == 0) return b2Vec2 (0, 0);

	massCenter / = boidsNum;
	float dist = Vec :: Distance (m_boids [j] -> GetBody () ->
GetWorldCenter (), massCenter);

	b2Vec2 vec = massCenter – m_boids [j] -> GetBody () ->
GetWorldCenter ();
	vec.Normalize ();

	return vec * dist;
}

b2Vec2 BoidManager :: MoveFromNeighbors (int j)
{
	float maxDist = 40;
	float neighborNum = 0;
	b2Vec2 vec (0, 0);

	for (int i = 0; i <m_boids.size (); ++ i)
	{
		if (i! = j && m_boids [i] -> GetBody () -> GetType () ==
b2_dynamicBody)
		{
			float dist = Vec :: Distance (m_boids [i] -> GetBody
() -> GetWorldCenter (), m_boids [j] -> GetBody () -> GetWorldCenter
());

			if (dist <maxDist)
			{
				b2Vec2 direction = m_boids [j] -> GetBody () -
> GetWorldCenter () – m_boids [i] -> GetBody () -> GetWorldCenter
();

				direction.Normalize ();
				direction * = (maxDist – dist);

				vec + = direction;
```

```
                    neighborNum ++;
                }
            }
        }

        return vec / (neighborNum! = 0? neighborNum: 1);
}

b2Vec2 BoidManager :: MoveFromObstacle (int j)
{
        float obsNum = 0;
        b2Vec2 vec (0, 0);

        for (int i = 0; i <m_obstacles.size (); ++ i)
        {
            b2AABB aabb = m_obstacles [i] -> GetFixture () -> GetAABB
(0);
            float size = aabb.upperBound.x – aabb.lowerBound.x;
            float dist = Vec :: Distance (m_boids [j] -> GetBody () -
> GetWorldCenter (), m_obstacles [i] -> GetBody () -> GetWorldCenter
());
            float maxDist = 20 + size * 2;

            if (dist <maxDist)
            {
                b2Vec2 direction = m_boids [j] -> GetBody () ->
GetWorldCenter () – m_obstacles [i] -> GetBody () -> GetWorldCenter
();

                direction.Normalize ();
                direction * = (maxDist – dist);

                vec + = direction;
                obsNum ++;
            }
        }

        return vec / (obsNum! = 0? obsNum: 1);
}

void BoidManager :: LimitSpeed (int j)
{
        b2Vec2 vel = m_boids [j] -> GetBody () -> GetLinearVelocity ();
        float speed = m_boids [j] -> m_speed / 2;

        if (vel.x> speed || vel.x <-speed || vel.y> speed || vel.y <-
speed)
        {
            vel.Normalize ();
            vel * = speed;
```

```
            m_boids [j] -> GetBody () -> SetLinearVelocity (vel);
      }
}

void BoidManager :: Draw (b2Vec2 mousePos)
{
      if (m_mouseTarget! = 0)
      {
            m_mouseTarget-> Draw (mousePos);
      }

      for (int i = 0; i <m_boids.size (); i ++)
      {
            g_debugDraw.DrawSolidCircle (m_boids [i] -> GetBody () ->
GetWorldCenter (), 0.3, b2Vec2 (0, 0), m_color);
      }

}

Boid * BoidManager :: FindBoidByBody (b2Body * body)
{
      for (int i = 0; i <m_boids.size (); i ++)
      {
            if (m_boids [i] -> GetBody () == body)
            {
                  return m_boids [i];
            }
      }

      return 0;
}
```

6) Energy.h

```
#ifndef ENERGY_H
#define ENERGY_H

#include "SimpleBody.h"

class Energy: public SimpleBody
{
public:

      Energy (b2World * world, bool dynamic, b2Vec2 pos, float size);
};

#endif

#include "Energy.h"
```

```
Energy :: Energy (b2World * world, bool dynamic, b2Vec2 pos, float
size): SimpleBody (world, dynamic, pos, size)
{
	m_energy = size * 20;

	m_body-> SetUserData ((void *) "energy");
}
```

7) EnergyManager.h

```
#ifndef ENERGY_MANAGER_H
#define ENERGY_MANAGER_H

#include "Energy.h"

class EnergyManager
{
public:

	EnergyManager ();

	void CreateEnergies (b2World * world, int groundWidth);
	void UpdateMouseTarget (b2Vec2 mousePos);
	void Draw (b2Vec2 mousePos);

	vector<Energy *> GetEnergies () {return m_energies; }

private:

	const int m_energyNum = 20;
	vector<Energy *> m_energies;

	SimpleBody * m_mouseTarget;
};

#endif

#include "EnergyManager.h"

EnergyManager :: EnergyManager ()
{
	m_mouseTarget = 0;
}

void EnergyManager :: CreateEnergies (b2World * world, int
groundWidth)
{
	for (int i = 0; i <m_energyNum; i ++)
	{
		int energySize = 200 / m_energyNum;
```

```
            b2Vec2 pos (-groundWidth + rand()% GroundWidth * 2, -
groundWidth + rand ()% groundWidth * 2);
            m_energies.push_back(New Energy (world, false, pos, 1 +
rand ()% energySize));
      }
}

void EnergyManager :: UpdateMouseTarget (b2Vec2 mousePos)
{
      m_mouseTarget = 0;

      for (int i = 0; i <m_energies.size (); i ++)
      {
            if (m_energies [i] -> GetFixture () -> TestPoint
(mousePos))
            {
                  m_mouseTarget = m_energies [i];
            }
      }
}

void EnergyManager :: Draw (b2Vec2 mousePos)
{
      if (m_mouseTarget! = 0)
      {
            m_mouseTarget-> Draw (mousePos);
      }
}
```

8) Obstacle.h

```
#ifndef OBSTACLE_H
#define OBSTACLE_H

#include <Box2D / Box2D.h>

class Obstacle
{
public:

      Obstacle (b2World * world, b2Vec2 pos, float size);

      b2Body * GetBody () {return m_body; }
      b2Fixture * GetFixture () {return m_fixture; }

protected:

      b2Body * m_body;
      b2Fixture * m_fixture;
};
```

```
#endif

#include "Obstacle.h"

Obstacle :: Obstacle (b2World * world, b2Vec2 pos, float size)
{
      b2BodyDef bd;
      bd.position.Set (pos.x, pos.y);
      m_body = world-> CreateBody (& bd);
      m_body-> SetUserData ((void *) "obstacle");

      b2PolygonShape shape;
      shape.SetAsBox (size, size);

      b2FixtureDef fd;
      fd.shape = & shape;
      fd.density = 1.f;
      fd.restitution = 0.2f;
      fd.friction = 1.f;
      m_fixture = m_body-> CreateFixture (& fd);
}
```

9) ObsManager.h

```
#ifndef OBS_MANAGER_H
#define OBS_MANAGER_H

#include <vector>
#include "Obstacle.h"

using namespace std;

class ObsManager
{
public:

      ObsManager ();

      void CreateEnergies (b2World * world, int groundWidth);

      vector<Obstacle *> GetObstacles () {return m_obstacles; }

private:

      const int m_obsCount = 20;
      vector<Obstacle *> m_obstacles;
};

#endif

#include "ObsManager.h"
```

```
ObsManager :: ObsManager ()
{

}

void ObsManager :: CreateEnergies (b2World * world, int groundWidth)
{
	for (int i = 0; i <m_obsCount; i ++)
	{
		int obsSize = 100 / m_obsCount;
		b2Vec2 pos (-groundWidth + rand()% GroundWidth * 2, -
groundWidth + rand ()% groundWidth * 2);
		m_obstacles.push_back(New Obstacle (world, pos, 1 + rand
()% obsSize));
	}
}
```

10) Spike.h

```
#ifndef SPIKE_H
#define SPIKE_H

#include <Box2D / Box2D.h>

class Spike
{
public:

	Spike (b2World * world, b2Vec2 boidPos, double boidSize, b2Vec2
mousePos);
	~ Spike ();

	void ApplyForce (b2Vec2 A, b2Vec2 B, double speed);

	void SetHit (bool isHit)
	{
		m_isHit = isHit;
	}

	bool IsHit ()
	{
		return m_isHit;
	}

	b2Body * GetBody ()
	{
		return m_body;
	}

	b2Fixture * GetFixture ()
```

```
	{
		return m_fixture;
	}

public:

	const double m_length = 0.3;
	const double m_speed = 100;
	const double m_energyStealing = 0.3;

protected:

	bool m_isHit;
	b2Body * m_body;
	b2Fixture * m_fixture;
};

#endif

#include "Spike.h"
#include "Vec.h"

Spike :: Spike (b2World * world, b2Vec2 boidPos, double boidSize,
b2Vec2 mousePos)
{
	m_isHit = false;

	b2BodyDef bd;
	bd.type = b2_dynamicBody;
	m_body = world-> CreateBody (& bd);

	m_body-> SetLinearDamping (0.5f);

	b2Vec2 vec1 = Vec :: Move (boidPos, mousePos, boidSize * 1.5);
	b2Vec2 vec2 = Vec :: Move (vec1, mousePos, m_length);

	b2EdgeShape shape;
	shape.Set (vec1, vec2);

	b2FixtureDef fd;
	fd.shape = & shape;
	fd.density = 0.1f;
	fd.restitution = 0.f;
	fd.friction = 1.f;
	m_fixture = m_body-> CreateFixture (& fd);

	ApplyForce (boidPos, mousePos, m_speed);

	m_body-> SetUserData ((void *) "spike");
}
```

```
Spike :: ~ Spike ()
{
      m_body-> GetWorld () -> DestroyBody (m_body);
}

void Spike :: ApplyForce (b2Vec2 A, b2Vec2 B, double speed)
{
      b2Vec2 vec = B – A;

      vec.Normalize ();
      vec * = speed;

      m_body-> ApplyLinearImpulse (vec, m_body-> GetWorldCenter (),
true);
}
```

11) SpileManager.h

```
#ifndef SPIKE_MANAGER_H
#define SPIKE_MANAGER_H

#include <vector>
#include "Spike.h"
#include "Boid.h"

using namespace std;

class RayCastCallback: public b2RayCastCallback
{
public:
      RayCastCallback ();

      float32 ReportFixture (b2Fixture * fixture, const b2Vec2 &
point, const b2Vec2 & normal, float32 fraction);

      bool m_hit;
      b2Vec2 m_point;
      b2Vec2 m_normal;
      char * m_enemyData;
};

class SpikeManager
{
public:

      void UpdateSpikes ();
      void AddSpike (b2World * world, vector<Boid *> boids, vector
<Boid *> enemyBoids);
      void DeleteSpikes ();
      Spike * FindSpikeByBody (b2Body * body);
```

```
	Boid * FindClosestBoid (Boid * boid, vector<Boid *>
enemyBoids);

private:

	vector<Spike *> m_spikes;
};

#endif

#include "SpikeManager.h"
#include "Vec.h"
#include "../../Framework/Test.h" // wtf omg

RayCastCallback :: RayCastCallback ()
{
	m_hit = false;
}

float32 RayCastCallback :: ReportFixture (b2Fixture * fixture, const
b2Vec2 & point, const b2Vec2 & normal, float32 fraction)
{
	b2Body * body = fixture-> GetBody ();
	const char * data = (const char *) body-> GetUserData ();

	if (strcmp(Data, m_enemyData)! = 0)
	{
		m_hit = true;
		m_point = point;
		m_normal = normal;
	}

	return 0;
}

void SpikeManager :: UpdateSpikes ()
{
	DeleteSpikes ();
}

void SpikeManager :: AddSpike (b2World * world, vector <Boid *>
boids, vector <Boid *> enemyBoids)
{
	for (int i = 0; i <boids.size (); i ++)
	{
		if (boids [i] -> GetBody () -> GetType () ==
b2_dynamicBody)
		{
			Boid * enemy = FindClosestBoid (boids [i],
enemyBoids);
```

```
                if (enemy! = 0)
                {
                      float boidSize = boids [i] -> GetFixture () ->
GetShape () -> m_radius;
                      b2Vec2 boidPos = boids [i] -> GetBody () ->
GetWorldCenter ();
                      b2Vec2 enemyPos = enemy-> GetBody () ->
GetWorldCenter ();

                      RayCastCallback callback;
                      callback.m_enemyData = (char *) enemy->
GetBody () -> GetUserData ();
                      world-> RayCast (& callback, boidPos,
enemyPos);

                      bool swarm = boids [i] -> GetParent () ->
IsSwarm ();
                      bool ok = boids [i] -> m_fieldView> Vec ::
Distance (boidPos, enemyPos);

                      if ((! callback.m_hit ||! boids [i] ->
GetParent () -> IsSwarm ()) && boids [i] -> m_fieldView> Vec ::
Distance (boidPos, enemyPos))
                      {
                            m_spikes.push_back(New Spike (world,
boidPos, boidSize, enemyPos));
                      }
                }
          }
     }
}

Boid * SpikeManager :: FindClosestBoid (Boid * boid, vector <Boid *>
enemyBoids)
{
     Boid * closest = 0;
     float minDist = FLT_MAX;

     for (int i = 0; i <enemyBoids.size (); i ++)
     {
          if (enemyBoids [i] -> GetBody () -> GetType () ==
b2_dynamicBody)
          {
                b2Vec2 boidPos = boid-> GetBody () -> GetWorldCenter
();
                b2Vec2 enemyPos = enemyBoids [i] -> GetBody () ->
GetWorldCenter ();
                float dist = Vec :: Distance (boidPos, enemyPos);

                if (minDist> dist)
                {
```

```
                    closest = enemyBoids [i];
                    minDist = dist;
                }
            }
        }

        return closest;
}

void SpikeManager :: DeleteSpikes ()
{
        for (vector<Spike *> :: iterator i = m_spikes.begin (); i! =
m_spikes.end ();)
        {
            if ((* i) -> IsHit () || (* i) -> GetBody () -> IsAwake
() == false)
            {
                delete * i;
                i = m_spikes.erase(I);
            }
            else
            {
                ++ i;
            }
        }
}

Spike * SpikeManager :: FindSpikeByBody (b2Body * body)
{
        for (int i = 0; i <m_spikes.size(); i ++)
        {
            if (m_spikes [i] -> GetBody () == body)
            {
                return m_spikes [i];
            }
        }

        return 0;
}
```

13) Test.h

```
#ifndef TEST_H
#define TEST_H

#include <Box2D / Box2D.h>
#include "DebugDraw.h"

#if defined (__ APPLE__)
#include <OpenGL / gl3.h>
#else
```

```
#include <glew / glew.h>
#endif
#include <glfw / glfw3.h>

#include <stdlib.h>

class Test;
struct Settings;

typedef Test * TestCreateFcn ();

#define    RAND_LIMIT 32767
#define DRAW_STRING_NEW_LINE 16

/// Random number in range [-1,1]
inline float32 RandomFloat ()
{
      float32 r = (float32) (std:: rand () & (RAND_LIMIT));
      r / = RAND_LIMIT;
      r = 2.0f * R – 1.0f;
      return r;
}

/// Random floating point number in range [lo, hi]
inline float32 RandomFloat (float32 lo, float32 hi)
{
      float32 r = (float32) (std:: rand () & (RAND_LIMIT));
      r / = RAND_LIMIT;
      r = (hi – lo) * r + lo;
      return r;
}

/// Test settings. Some can be controlled in the GUI.
struct Settings
{
      Settings ()
      {
            hz = 60.0f;
            velocityIterations = 8;
            positionIterations = 3;
            drawShapes = true;
            drawJoints = true;
            drawAABBs = false;
            drawContactPoints = false;
            drawContactNormals = false;
            drawContactImpulse = false;
            drawFrictionImpulse = false;
            drawCOMs = false;
            drawStats = false;
            drawProfile = false;
            enableWarmStarting = true;
```

```
            enableContinuous = true;
            enableSubStepping = false;
            enableSleep = true;
            pause = false;
            singleStep = false;
      }

      float32 hz;
      int32 velocityIterations;
      int32 positionIterations;
      bool drawShapes;
      bool drawJoints;
      bool drawAABBs;
      bool drawContactPoints;
      bool drawContactNormals;
      bool drawContactImpulse;
      bool drawFrictionImpulse;
      bool drawCOMs;
      bool drawStats;
      bool drawProfile;
      bool enableWarmStarting;
      bool enableContinuous;
      bool enableSubStepping;
      bool enableSleep;
      bool pause;
      bool singleStep;
};

struct TestEntry
{
      const char * name;
      TestCreateFcn * createFcn;
};

extern TestEntry g_testEntries [];
// This is called when a joint in the world is implicitly destroyed
// because an attached body is destroyed. This gives us a chance to
// nullify the mouse joint.
class DestructionListener: public b2DestructionListener
{
public:
      void SayGoodbye (b2Fixture * fixture) {B2_NOT_USED (fixture); }
      void SayGoodbye (b2Joint * joint);

      Test * test;
};

const int32 k_maxContactPoints = 2048;

struct ContactPoint
{
```

```
	b2Fixture * fixtureA;
	b2Fixture * fixtureB;
	b2Vec2 normal;
	b2Vec2 position;
	b2PointState state;
	float32 normalImpulse;
	float32 tangentImpulse;
	float32 separation;
};

class Test: public b2ContactListener
{
public:

	Test ();
	virtual ~ Test ();

	void DrawTitle (const char * string);
	virtual void Step (Settings * settings);
	virtual void Keyboard (int key) {B2_NOT_USED (key); }
	virtual void KeyboardUp (int key) {B2_NOT_USED (key); }
	void ShiftMouseDown (const b2Vec2 & p);
	virtual void MouseDown (const b2Vec2 & p);
	virtual void MouseUp (const b2Vec2 & p);
	virtual void MouseMove (const b2Vec2 & p);
	void LaunchBomb ();
	void LaunchBomb (const b2Vec2 & position, const b2Vec2 &
velocity);

	void SpawnBomb (const b2Vec2 & worldPt);
	void CompleteBombSpawn (const b2Vec2 & p);

	// Let derived tests know that a joint was destroyed.
	virtual void JointDestroyed (b2Joint * joint) {B2_NOT_USED
(joint); }

	// Callbacks for derived classes.
	virtual void BeginContact (b2Contact * contact) {B2_NOT_USED
(contact); }
	virtual void EndContact (b2Contact * contact) {B2_NOT_USED
(contact); }
	virtual void PreSolve (b2Contact * contact, const b2Manifold *
oldManifold);
	virtual void PostSolve (b2Contact * contact, const
b2ContactImpulse * impulse)
	{
		B2_NOT_USED (contact);
		B2_NOT_USED (impulse);
	}

	void ShiftOrigin (const b2Vec2 & newOrigin);
```

```
protected:
      friend class DestructionListener;
      friend class BoundaryListener;
      friend class ContactListener;

      b2Body * m_groundBody;
      b2AABB m_worldAABB;
      ContactPoint m_points [k_maxContactPoints];
      int32 m_pointCount;
      DestructionListener m_destructionListener;
      int32 m_textLine;
      b2World * m_world;
      b2Body * m_bomb;
      b2MouseJoint * m_mouseJoint;
      b2Vec2 m_bombSpawnPoint;
      bool m_bombSpawning;
      b2Vec2 m_mouseWorld;
      int32 m_stepCount;

      b2Profile m_maxProfile;
      b2Profile m_totalProfile;
};

#endif

#include "Test.h"
#include <stdio.h>

void DestructionListener :: SayGoodbye (b2Joint * joint)
{
      if (test-> m_mouseJoint == joint)
      {
            test-> m_mouseJoint = NULL;
      }
      else
      {
            test-> JointDestroyed (joint);
      }
}

Test :: Test ()
{
      b2Vec2 gravity;
      gravity.Set (0.0f, 0.0f);
      m_world = new b2World (gravity);
      m_bomb = NULL;
      m_textLine = 30;
      m_mouseJoint = NULL;
      m_pointCount = 0;
```

```
	m_destructionListener.test = this;
	m_world-> SetDestructionListener (& m_destructionListener);
	m_world-> SetContactListener (this);
	m_world-> SetDebugDraw (& g_debugDraw);

	m_bombSpawning = false;

	m_stepCount = 0;

	b2BodyDef bodyDef;
	m_groundBody = m_world-> CreateBody (& bodyDef);

	memset(& M_maxProfile, 0, sizeof (b2Profile));
	memset(& M_totalProfile, 0, sizeof (b2Profile));
}

Test :: ~ Test ()
{
	// By deleting the world, we delete the bomb, mouse joint, etc.
	delete m_world;
	m_world = NULL;
}

void Test :: PreSolve (b2Contact * contact, const b2Manifold *
oldManifold)
{
	const b2Manifold * manifold = contact-> GetManifold ();

	if (manifold-> pointCount == 0)
	{
		return;
	}

	b2Fixture * fixtureA = contact-> GetFixtureA ();
	b2Fixture * fixtureB = contact-> GetFixtureB ();

	b2PointState state1 [b2_maxManifoldPoints], state2
[b2_maxManifoldPoints];
	b2GetPointStates (state1, state2, oldManifold, manifold);

	b2WorldManifold worldManifold;
	contact-> GetWorldManifold (& worldManifold);

	for (int32 i = 0; i <manifold-> pointCount && m_pointCount
<k_maxContactPoints; ++ i)
	{
		ContactPoint * cp = m_points + m_pointCount;
		cp-> fixtureA = fixtureA;
		cp-> fixtureB = fixtureB;
		cp-> position = worldManifold.points [i];
		cp-> normal = worldManifold.normal;
```

```
            cp-> state = state2 [i];
            cp-> normalImpulse = manifold-> points [i]
.normalImpulse;
            cp-> tangentImpulse = manifold-> points [i]
.tangentImpulse;
            cp-> separation = worldManifold.separations [i];
            ++ m_pointCount;
      }
}

void Test :: DrawTitle (const char * string)
{
 g_debugDraw.DrawString (5, DRAW_STRING_NEW_LINE, string);
 m_textLine = 3 * DRAW_STRING_NEW_LINE;
}

class QueryCallback: public b2QueryCallback
{
public:
      QueryCallback (const b2Vec2 & point)
      {
            m_point = point;
            m_fixture = NULL;
      }

      bool ReportFixture (b2Fixture * fixture)
      {
            b2Body * body = fixture-> GetBody ();
            if (body-> GetType () == b2_dynamicBody)
            {
                  bool inside = fixture-> TestPoint (m_point);
                  if (inside)
                  {
                        m_fixture = fixture;

                        // We are done, terminate the query.
                        return false;
                  }
            }

            // Continue the query.
            return true;
      }

      b2Vec2 m_point;
      b2Fixture * m_fixture;
};

void Test :: MouseDown (const b2Vec2 & p)
{
      m_mouseWorld = p;
```

```
	if (m_mouseJoint! = NULL)
	{
		return;
	}

	// Make a small box.
	b2AABB aabb;
	b2Vec2 d;
	d.Set (0.001f, 0.001f);
	aabb.lowerBound = p – d;
	aabb.upperBound = p + d;

	// Query the world for overlapping shapes.
	QueryCallback callback (p);
	m_world-> QueryAABB (& callback, aabb);

	if (callback.m_fixture)
	{
		b2Body * body = callback.m_fixture-> GetBody ();
		b2MouseJointDef md;
		md.bodyA = m_groundBody;
		md.bodyB = body;
		md.target = p;
		md.maxForce = 1000.0f * Body-> GetMass ();
		m_mouseJoint = (b2MouseJoint *) m_world-> CreateJoint (&
md);
		body-> SetAwake (true);
	}
}

void Test :: SpawnBomb (const b2Vec2 & worldPt)
{
	m_bombSpawnPoint = worldPt;
	m_bombSpawning = true;
}

void Test :: CompleteBombSpawn (const b2Vec2 & p)
{
	if (m_bombSpawning == false)
	{
		return;
	}

	const float multiplier = 30.0f;
	b2Vec2 vel = m_bombSpawnPoint – p;
	vel * = multiplier;
	LaunchBomb (m_bombSpawnPoint, vel);
	m_bombSpawning = false;
}
```

```
void Test :: ShiftMouseDown (const b2Vec2 & p)
{
	m_mouseWorld = p;

	if (m_mouseJoint! = NULL)
	{
		return;
	}

	SpawnBomb (p);
}

void Test :: MouseUp (const b2Vec2 & p)
{
	if (m_mouseJoint)
	{
		m_world-> DestroyJoint (m_mouseJoint);
		m_mouseJoint = NULL;
	}

	if (m_bombSpawning)
	{
		CompleteBombSpawn (p);
	}
}

void Test :: MouseMove (const b2Vec2 & p)
{
	m_mouseWorld = p;

	if (m_mouseJoint)
	{
		m_mouseJoint-> SetTarget (p);
	}
}

void Test :: LaunchBomb ()
{
	b2Vec2 p (RandomFloat (-15.0f, 15.0f), 30.0f);
	b2Vec2 v = -5.0f * P;
	LaunchBomb (p, v);
}

void Test :: LaunchBomb (const b2Vec2 & position, const b2Vec2 &
velocity)
{
	if (m_bomb)
	{
		m_world-> DestroyBody (m_bomb);
		m_bomb = NULL;
	}
```

```
	b2BodyDef bd;
	bd.type = b2_dynamicBody;
	bd.position = position;
	bd.bullet = true;
	m_bomb = m_world-> CreateBody (& bd);
	m_bomb-> SetLinearVelocity (velocity);

	b2CircleShape circle;
	circle.m_radius = 0.3f;

	b2FixtureDef fd;
	fd.shape = & circle;
	fd.density = 20.0f;
	fd.restitution = 0.0f;

	b2Vec2 minV = position – b2Vec2 (0.3f, 0.3f);
	b2Vec2 maxV = position + b2Vec2 (0.3f, 0.3f);

	b2AABB aabb;
	aabb.lowerBound = minV;
	aabb.upperBound = maxV;

	m_bomb-> CreateFixture (& fd);
}

void Test :: Step (Settings * settings)
{
	float32 timeStep = settings-> hz> 0.0f ? 1.0f / Settings-> hz:
float32 (0.0f);

	if (settings-> pause)
	{
		if (settings-> singleStep)
		{
			settings-> singleStep = 0;
		}
		else
		{
			timeStep = 0.0f;
		}

		g_debugDraw.DrawString (5, m_textLine, "**** PAUSED
****");
		m_textLine + = DRAW_STRING_NEW_LINE;
	}

	uint32 flags = 0;
	flags + = settings-> drawShapes            * B2Draw ::
e_shapeBit;
```

```
      flags + = settings-> drawJoints            * B2Draw ::
e_jointBit;
      flags + = settings-> drawAABBs             * B2Draw ::
e_aabbBit;
      flags + = settings-> drawCOMs                    * B2Draw ::
e_centerOfMassBit;
      g_debugDraw.SetFlags (flags);

      m_world-> SetAllowSleeping (settings-> enableSleep);
      m_world-> SetWarmStarting (settings-> enableWarmStarting);
      m_world-> SetContinuousPhysics (settings-> enableContinuous);
      m_world-> SetSubStepping (settings-> enableSubStepping);

      m_pointCount = 0;

      m_world-> Step (timeStep, settings-> velocityIterations,
settings-> positionIterations);

      m_world-> DrawDebugData ();
 g_debugDraw.Flush ();

      if (timeStep> 0.0f)
      {
            ++ m_stepCount;
      }

      if (settings-> drawStats)
      {
            int32 bodyCount = m_world-> GetBodyCount ();
            int32 contactCount = m_world-> GetContactCount ();
            int32 jointCount = m_world-> GetJointCount ();
            g_debugDraw.DrawString (5, m_textLine, "bodies / contacts
/ joints =% d /% d /% d", bodyCount, contactCount, jointCount);
            m_textLine + = DRAW_STRING_NEW_LINE;

            int32 proxyCount = m_world-> GetProxyCount ();
            int32 height = m_world-> GetTreeHeight ();
            int32 balance = m_world-> GetTreeBalance ();
            float32 quality = m_world-> GetTreeQuality ();
            g_debugDraw.DrawString (5, m_textLine, "proxies / height
/ balance / quality =% d /% d /% d /% g", proxyCount, height,
balance, quality);
            m_textLine + = DRAW_STRING_NEW_LINE;
      }

      // Track maximum profile times
      {
            const b2Profile & p = m_world-> GetProfile ();
            m_maxProfile.step = b2Max (m_maxProfile.step, p.step);
            m_maxProfile.collide = b2Max (m_maxProfile.collide,
p.collide);
```

```
            m_maxProfile.solve = b2Max (m_maxProfile.solve, p.solve);
            m_maxProfile.solveInit = b2Max (m_maxProfile.solveInit,
p.solveInit);
            m_maxProfile.solveVelocity = b2Max
(m_maxProfile.solveVelocity, p.solveVelocity);
            m_maxProfile.solvePosition = b2Max
(m_maxProfile.solvePosition, p.solvePosition);
            m_maxProfile.solveTOI = b2Max (m_maxProfile.solveTOI,
p.solveTOI);
            m_maxProfile.broadphase = b2Max (m_maxProfile.broadphase,
p.broadphase);

            m_totalProfile.step + = p.step;
            m_totalProfile.collide + = p.collide;
            m_totalProfile.solve + = p.solve;
            m_totalProfile.solveInit + = p.solveInit;
            m_totalProfile.solveVelocity + = p.solveVelocity;
            m_totalProfile.solvePosition + = p.solvePosition;
            m_totalProfile.solveTOI + = p.solveTOI;
            m_totalProfile.broadphase + = p.broadphase;
      }

      if (settings-> drawProfile)
      {
            const b2Profile & p = m_world-> GetProfile ();

            b2Profile aveProfile;
            memset(& AveProfile, 0, sizeof (b2Profile));
            if (m_stepCount> 0)
            {
                  float32 scale = 1.0f / M_stepCount;
                  aveProfile.step = scale * m_totalProfile.step;
                  aveProfile.collide = scale * m_totalProfile.collide;
                  aveProfile.solve = scale * m_totalProfile.solve;
                  aveProfile.solveInit = scale *
m_totalProfile.solveInit;
                  aveProfile.solveVelocity = scale *
m_totalProfile.solveVelocity;
                  aveProfile.solvePosition = scale *
m_totalProfile.solvePosition;
                  aveProfile.solveTOI = scale *
m_totalProfile.solveTOI;
                  aveProfile.broadphase = scale *
m_totalProfile.broadphase;
            }

            g_debugDraw.DrawString (5, m_textLine, "step [ave] (max)
=% 5.2f [% 6.2f] (% 6.2f)", p.step, aveProfile.step,
m_maxProfile.step);
            m_textLine + = DRAW_STRING_NEW_LINE;
```

```
            g_debugDraw.DrawString (5, m_textLine, "collide [ave]
(max) =% 5.2f [% 6.2f] (% 6.2f)", p.collide, aveProfile.collide,
m_maxProfile.collide);
            m_textLine + = DRAW_STRING_NEW_LINE;
            g_debugDraw.DrawString (5, m_textLine, "solve [ave] (max)
=% 5.2f [% 6.2f] (% 6.2f)", p.solve, aveProfile.solve,
m_maxProfile.solve);
            m_textLine + = DRAW_STRING_NEW_LINE;
            g_debugDraw.DrawString (5, m_textLine, "solve init [ave]
(max) =% 5.2f [% 6.2f] (% 6.2f)", p.solveInit, aveProfile.solveInit,
m_maxProfile.solveInit);
            m_textLine + = DRAW_STRING_NEW_LINE;
            g_debugDraw.DrawString (5, m_textLine, "solve velocity
[ave] (max) =% 5.2f [% 6.2f] (% 6.2f)", p.solveVelocity,
aveProfile.solveVelocity, m_maxProfile.solveVelocity);
            m_textLine + = DRAW_STRING_NEW_LINE;
            g_debugDraw.DrawString (5, m_textLine, "solve position
[ave] (max) =% 5.2f [% 6.2f] (% 6.2f)", p.solvePosition,
aveProfile.solvePosition, m_maxProfile.solvePosition);
            m_textLine + = DRAW_STRING_NEW_LINE;
            g_debugDraw.DrawString (5, m_textLine, "solveTOI [ave]
(max) =% 5.2f [% 6.2f] (% 6.2f)", p.solveTOI, aveProfile.solveTOI,
m_maxProfile.solveTOI);
            m_textLine + = DRAW_STRING_NEW_LINE;
            g_debugDraw.DrawString (5, m_textLine, "broad-phase [ave]
(max) =% 5.2f [% 6.2f] (% 6.2f)", p.broadphase,
aveProfile.broadphase, m_maxProfile.broadphase);
            m_textLine + = DRAW_STRING_NEW_LINE;
      }

      if (m_mouseJoint)
      {
            b2Vec2 p1 = m_mouseJoint-> GetAnchorB ();
            b2Vec2 p2 = m_mouseJoint-> GetTarget ();

            b2Color c;
            c.Set (0.0f, 1.0f, 0.0f);
            g_debugDraw.DrawPoint (p1, 4.0f, c);
            g_debugDraw.DrawPoint (p2, 4.0f, c);

            c.Set (0.8f, 0.8f, 0.8f);
            g_debugDraw.DrawSegment (p1, p2, c);
      }

      if (m_bombSpawning)
      {
            b2Color c;
            c.Set (0.0f, 0.0f, 1.0f);
            g_debugDraw.DrawPoint (m_bombSpawnPoint, 4.0f, c);

            c.Set (0.8f, 0.8f, 0.8f);
```

```
            g_debugDraw.DrawSegment (m_mouseWorld, m_bombSpawnPoint,
c);
      }

      if (settings-> drawContactPoints)
      {
            const float32 k_impulseScale = 0.1f;
            const float32 k_axisScale = 0.3f;

            for (int32 i = 0; i <m_pointCount; ++ i)
            {
                  ContactPoint * point = m_points + i;

                  if (point-> state == b2_addState)
                  {
                        // Add
                        g_debugDraw.DrawPoint (point-> position,
10.0f, b2Color (0.3f, 0.95f, 0.3f));
                  }
                  else if (point-> state == b2_persistState)
                  {
                        // Persist
                        g_debugDraw.DrawPoint (point-> position, 5.0f,
b2Color (0.3f, 0.3f, 0.95f));
                  }

                  if (settings-> drawContactNormals == 1)
                  {
                        b2Vec2 p1 = point-> position;
                        b2Vec2 p2 = p1 + k_axisScale * point-> normal;
                        g_debugDraw.DrawSegment (p1, p2, b2Color
(0.9f, 0.9f, 0.9f));
                  }
                  else if (settings-> drawContactImpulse == 1)
                  {
                        b2Vec2 p1 = point-> position;
                        b2Vec2 p2 = p1 + k_impulseScale * point->
normalImpulse * point-> normal;
                        g_debugDraw.DrawSegment (p1, p2, b2Color
(0.9f, 0.9f, 0.3f));
                  }

                  if (settings-> drawFrictionImpulse == 1)
                  {
                        b2Vec2 tangent = b2Cross (point-> normal,
1.0f);
                        b2Vec2 p1 = point-> position;
                        b2Vec2 p2 = p1 + k_impulseScale * point->
tangentImpulse * tangent;
                        g_debugDraw.DrawSegment (p1, p2, b2Color
(0.9f, 0.9f, 0.3f));
```

```
                }
            }
        }
}

void Test :: ShiftOrigin (const b2Vec2 & newOrigin)
{
      m_world-> ShiftOrigin (newOrigin);
}
```

ДОДАТОК Б

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	123 с.
	Інші документи	
2.	Презентаційні матеріали	__ плакатів

		Прізвище та ініціали.	Підп.	Дата			
Розробив		Верхогляд І.О.			Дослідження методів ройового інтелекту і агентних систем	Шифр групи	Код напр./спец.
Перевірив		Золотухін О.В.				СШІм-19-2	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Філатов В.О.					