Methodes and Techniques report
Huub Spekman
500639106
Team namespace

Table of Contents

Introduction

The project game technology in year 3 of the HvA tries to teach us more about C++, 3D and game engines. The assignment is clear, create a game in C++, 3D and you can choose from the engines Irrlicht, Cocos3D or Ogre3D. There are some technical requirements aswell. We chose Ogre3D as our engine, this report will explain how well I did in each of the Method and Techniques assesment criteria.

C++ 9 points

In order to qualify for 9 points I have to show language specific concepts, including namespaces, inheritance and advanced concept. My advanced concept is a template class I made for the QuadTree, which also serves as my advanced technique.

Language specific concepts

Namespaces
We use the namespace Ogre for all Ogre Objects in our games. Examples are Vectors, Nodes, Overlaysystem, Scenemanger and camera. We use the namespace OIS for input/output system Ogre provides us With. Examples are the Keyboard object and the InputManager. We also use the std namespace for vectors and the OgreOggsound namespace for the soundmanager. We dont use namespaces for our game itself since we didnt really see any advantages in using them.

Example given, the namespaces used in our main application class (with unneccary code for this example replaced with ...)

```
class Application
{
	...


	Ogre::Root* mRoot;
	Ogre::String mResourcesConfig;
	Ogre::String mPluginConfig;
	Ogre::OverlaySystem* mOverlaySystem;
	Ogre::SceneManager* mSceneManager;
	OgreOggSound::OgreOggSoundManager* mSoundManager;
	Ogre::RenderWindow* mWindow;
	Ogre::Camera* mCamera;

	OIS::InputManager* mInputManager;
	OIS::Mouse* mMouse;
	OIS::Keyboard* mKeyboard;

	Integrator* mIntegrator;

	InputManager* _inputManager; // TODO: change name to prevent confusion
(InputControler?)
	std::array<Player*, 4> players;
	std::array<Ogre::TextAreaOverlayElement*, 4> textAreas;
	std::vector<GameObject*> gameObjects;
	std::vector<AABB*> boundingBoxes;
	QuadTree<GameObject*>* quadTree;


	...
};
```

Inheritance

I use inheritance for our GameObjects, GameObjects has two child classes, Player and Prop. With all Players being able to be controlled by a user and all Props being stationary part of the environment. We use inheritance here because both classes share a lot of logic, both need to be updated every frame, both need to be rendered, both have boundingboxes, etcetera. However they also have a fair share of individual logic, if they are controlled by input, if they can be picked up etcetera. This balance between shared and individual logic made it a logical choice to use inheritance. I also used inheritance for our own input manager, this class uses the Ogre::KeyListener class so it can react to keyboard inputs.

Base Class GameObject:

```
class GameObject
{
	private:

	protected:
		Ogre::String name;
		bool _alive;
		bool _physics;

	public:
		bool isPickedUp;
		float weight;
		float bounciness;
		Ogre::Vector3* pickedUpPosition;

		Rigidbody* body;
		Ogre::SceneNode* node;
		AABB* boundingbox;

		GameObject();
		GameObject(Ogre::Vector3, Ogre::SceneNode*);
		~GameObject();

		Ogre::String getName(){ return name; };
		bool isAlive(){ return _alive; };
		void kill();

		void handleCollision(GameObject* other, bool);

		bool isAffectedByPhysics(){ return _physics; };
		void setAffectedByPysics(bool physics) { _physics = physics; };

		virtual void impact(GameObject* other);
		virtual void update(const double&, const double&);
};
```

SubClasses Player and Prop:

```
class Prop : public GameObject
{
	private:
		OgreOggSound::OgreOggSoundManager* ppSoundManager;
	public:
		Prop();
		Prop(Ogre::Vector3, Ogre::SceneNode*);
		~Prop();

		Ogre::String matName;

		void update(const double&, const double&);
		void impact(GameObject* other);
};

class Player : public GameObject
{
	private:
		OgreOggSound::OgreOggSoundManager* pSoundManager;
		Prop* _highlightProp;
	public:
		int playerNumber;
		int lives;
		float smashPercentage;

		Player();
		Player(Ogre::Vector3, Ogre::SceneNode*);
		~Player();

		Ogre::Vector4 color;
		bool movesleft, movesright;
		bool pickedUpProp, pickingUpProp;
		GameObject* pickedUpPropptr;
		void updateMatAlpha();
		void setPlayerNumber(int playerNumber);
		void handleCollision(GameObject* other, bool);
		void impact(GameObject* other);
		void throwProp(float velocityX);

		// Actions
		void move(const Ogre::Vector3& direction);
		void jump();

		void respawn();
		void update(const double&, const double&);
};
```

Pointers
I use pointers a great deal in my code. The reason for this is that it is a quick and efficiant way to access the same objects from different places. My inputcontroller uses a pointer to the players they are controlling. the Boundingboxes use pointers to the objects they are linked to so they can handle collision. The objects have pointers to the bounding boxes since after refactoring the collision code we no longer keep a list of bounding boxes and we access the bounding boxes this way.

I did not use any operator overloading since there seemed allmost no benefit in using them in our game's architecture.

Advanced concept, Template class
I use a template class for the quadtree since this code seems usefull in future projects. By using a template class we can easily reuse this code after some renaming of Template members. We use the template class for our quadtree, With our current quad tree we look for our own GameObjects position members. However since we use a template class if we are to reuse this code in another project we can use whatever Data structure we want.

QuadTree class:

```
/*
Template class so we can reuse the code for OGRE projects that use different gameobject.
Ofcourse, some code would have to be changed, but only the naming.

the point of this class is to divide the field in quads so we can check for collision
more efficiant
We decided to use a QuadTree because our collision detection was very bad, inefficant
and had problems when deleting objects
*/
template <class T>
class QuadTree
{
private:
	// four quadtrees to split the current one
	QuadTree* _nw;
	QuadTree* _ne;
	QuadTree* _se;
	QuadTree* _sw;

	// array filled objects that are in the current quad
	std::vector<T> _objectArray;

	// lower left point and upper right point of current quad
	Ogre::Vector3 _minVector, _maxVector;
	bool _hasChildren;

	// returns area of the current quad, we use this to check if we need smaller
quads
	float getArea();
public:
	QuadTree();
	QuadTree(Ogre::Vector3 minVector, Ogre::Vector3 maxVector, std::vector<T>
objectArray);
	~QuadTree();
};
```

Tooling 8 points

Explanation of basic git concepts

GIT is a version control program. It is very usefull in case some code brakes. You can look back, see what changed and quickly identify what went wrong. It is also great to go back to a previous version to see how much has changed over time.

The best thing about GIT is that while the version control is all managed locally, you can add a “remote” to your GIT. This allows you to have identical projects stored on a server. This is usefull since you don’t lose anything if your computer stops working. Another added bonus of storing the GIT in an online repository is that it makes working toghether much easier, you can both push to the same server and than pull each others work. GIT has a built-in system that checks if there are conflicts between you and your colleagues code. If there is, it first tries to resolve itself however, if it cannot solve itself It will give you an overview of both versions and lets you decide how to resolve it.

I will now explain some git terms used and some git commands you can use.

Branch
A branch is a deviation of the current commit. Used mostly to work on different features.

GIT Add
When there are files you have changed you have to ‘add’ them so git knows which files you want to commit.

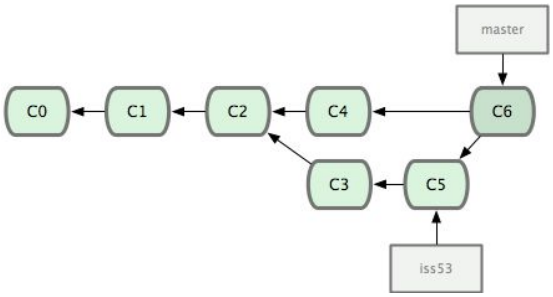
GIT Commit
This tells git to commit the current changes you have added. You also can set a message to remind yourself what changed in this specific commit.

GIT Fetch
This gets the changes made on the remote server but doesn’t change anything to your current commits.

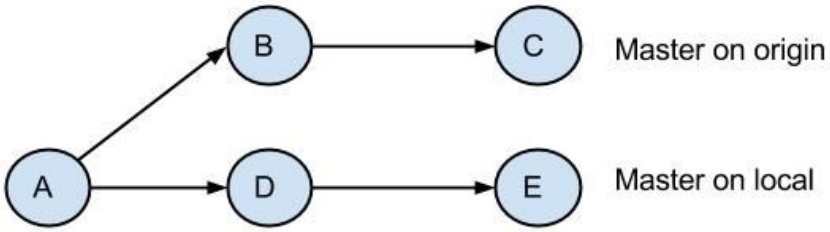
GIT Pull
This pulls all changes on the current branch and merges it to the current branch.

GIT Push
Pushes your current work to the remote. You can only push if your current commit is ahead of the commit found on the remote server.

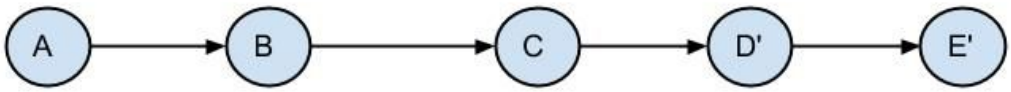
GIT Merge
Merges two commits that were branched out in different ways, git automatically checkes the differences and allows you to handle the conflicts in which way you want.



GIT Rebase
Rebases a commit to another commit that branched out. It has the same behaviour as a merge commit but the history looks different. This is more natural for smaller commits that don't impact or conflict with each other.



Before git pull --rebase



After git pull --rebase

How we used git

We had a couple of rules while using git this project. Whenever you started working on a new feature you had to start working in a new branch. When merging your current branch with another branch to work you allways sat down with the person who had created the code on the other branch to make sure there were no issues after the merge.

Asset import tool

We also used a tool to import our own assets, you can see the "tooling.pdf" file for more information on how we did that.

Coding standards 6 points

We created coding standards at the start of the projects. You can find them in the “Coding Standards.pdf” file also present at this git repository. From the code examples displayed elsewhere you can see we enforced it throughout the project. Since most conventions are about the header file I will show examples of a few header files that keep to these coding standards

```
class InputManager : public OIS::KeyListener
{
private:
	virtual bool keyPressed(const OIS::KeyEvent& ke);
	virtual bool keyReleased(const OIS::KeyEvent& ke);
	Player* _player;
	XInputManager* _XInputManager;
	OgreOggSound::OgreOggSoundManager* iSoundManager;
public:
	InputManager();
	InputManager(GameObject* player);
	InputManager(XInputManager* xim);
	~InputManager();
	std::array<Player*,4> players;
	void InputHandler();
	void setPlayer(GameObject* player, int i);
	void update(const double&, const double&);
};
```

```
class Player : public GameObject
{
	private:
		OgreOggSound::OgreOggSoundManager* _pSoundManager;
		Prop* _highlightProp;
	public:
		int playerNumber;
		int lives;
		float smashPercentage;

		Player();
		Player(Ogre::Vector3, Ogre::SceneNode*);
		~Player();

		Ogre::Vector4 color;
		bool movesleft, movesright;
		bool pickedUpProp, pickingUpProp;
		GameObject* pickedUpPropptr;
		void updateMatAlpha();
		void setPlayerNumber(int playerNumber);
		void handleCollision(GameObject* other, bool);
		void impact(GameObject* other);
		void throwProp(float velocityX);

		// Actions
		void move(const Ogre::Vector3& direction);
		void jump();

		void respawn();
		void update(const double&, const double&);
};
```

```
class Prop : public GameObject
{
	private:
		OgreOggSound::OgreOggSoundManager* _ppSoundManager;
	public:
		Prop();
		Prop(Ogre::Vector3, Ogre::SceneNode*);
		~Prop();

		Ogre::String matName;

		void update(const double&, const double&);
		void impact(GameObject* other);
};
```

```
template <class T>
class QuadTree
{
private:
	// four quadtrees to split the current one
	QuadTree* _nw;
	QuadTree* _ne;
	QuadTree* _se;
	QuadTree* _sw;

	// array filled objects that are in the current quad
	std::vector<T> _objectArray;

	// lower left point and upper right point of current quad
	Ogre::Vector3 _minVector, _maxVector;
	bool _hasChildren;

	// returns area of the current quad, we use this to check if we need smaller
quads
	float getArea();
public:
	QuadTree();
	QuadTree(Ogre::Vector3 minVector, Ogre::Vector3 maxVector, std::vector<T>
objectArray);
	~QuadTree();
};
```

Testing 2 points

I am the one who made our testing framework work as intended but I didn't have/take the time to write some unit tests so I can't really get many points for this. We use google test as a framework. It is a really nice framework that lets you test your code quite easily. This is an example of an test (not written by me)

```
TEST(Initialization, PlayerSpawn)
{
	ASSERT_EQ(app.players.size(), 4) << "There should be four player";
	ASSERT_TRUE(app.players[0]) << "player 0 isn't initialized";
	ASSERT_TRUE(app.players[1]) << "player 1 isn't initialized";
	ASSERT_TRUE(app.players[2]) << "player 2 isn't initialized";
	ASSERT_TRUE(app.players[3]) << "player 3 isn't initialized";
}
```

This is fired right after initialization, if it fails it tells us exactly which test fails with our own error code making it really easy to debug.

porting 0 points

We didn’t do any porting.

Documentation 5 points

I documented most of my code, however the level of documentation differs between different blocks of code. I documented the QuadTree class pretty in-depth but the rest of my code is pretty poorly documented, which is why I believe I only deserve 5 points for this category.

All QuadTree comments (screenshots because the code itself is too big to show)

```
/*
Template class so we can reuse the code for OGRE projects that use different gameobject.
Ofcourse, some code would have to be changed, but only the naming.

the point of this class is to divide the field in quads so we can check for collision more efficiant
We decided to use a QuadTree because our collision detection was very bad, inefficant and had problems when deleting objects
*/
template <class T>
class QuadTree
{
private:
    // four quadtrees to split the current one
    QuadTree* _nw;
    QuadTree* _ne;
    QuadTree* _se;
    QuadTree* _sw;

    // array filled objects that are in the current quad
    std::vector<T> _objectArray;

    // lower left point and upper right point of current quad
    Ogre::Vector3 _minVector, _maxVector;
    bool _hasChildren;

    // returns area of the current quad, we use this to check if we need smaller quads
    float getArea();
```

```
// Constructor, make the current quad and if there are more than 1 objects in the objectArray subdivide this quad into 4 new quads
template <class T>
QuadTree<T>::QuadTree(Ogre::Vector3 minVector, Ogre::Vector3 maxVector, std::vector<T> objectArray)
{
    _objectArray = objectArray;
    _ne = nullptr;
    _nw = nullptr;
    _se = nullptr;
    _sw = nullptr;
    _minVector = minVector;
    _maxVector = maxVector;

    // if there is only one object in this quad there is no reason to continue since there will be no collision detection needed
    if (objectArray.size() > 1){
        /* if the area of our current quad is smaller than 40000 we check the remaining objects in our array for collision
        * the reason our smallest quadtree has an area of 40000 is because our blocks are 50x50.
        * After testing we found that going below a 200x200 area will often result in having to check all these objects for collision anyway.
        * This happens because their boundingbox ends up in more than one child quad
        */
        if (getArea() < 40000){
            auto it = objectArray.begin();
            while (it != objectArray.end())
            {
                auto it2 = it + 1;
                while (it2 != objectArray.end()){
                    (*it)->boundingbox->isCollidingWithAABB((*it2)->boundingbox);
                    ++it2;
                }
                ++it;
            }
            return;
        }
        // make a vector in the middle of our quad so that we can devide this quad into 4 new ones
        Ogre::Vector3 middleVector = (minVector + maxVector) / 2;
        // make 4 new objectarrays, fill them and push them to the child quadtrees
        std::vector<T> ne, nw, se, sw;
```

getNormalVector code (not very well documented)

```
Ogre::Vector3 AABB::getNormalVector(AABB* player, AABB* prop){
	// calculate the normal vector
	Ogre::Vector3 differencevector = player->obj->body->position -
prop->obj->body->position;
	Ogre::Real floor, ceiling, leftwall, rightwall, smallestvalue;
	floor = differencevector.y - (prop->height / 2 + player->height / 2);
	ceiling = differencevector.y + (prop->height / 2 + player->height / 2);
	rightwall = differencevector.x - (prop->width / 2 + player->width / 2);
	leftwall = differencevector.x + (prop->width / 2 + player->width / 2);
	if (floor < 0) floor *= -1;
	if (ceiling < 0) ceiling *= -1;
	if (rightwall < 0) rightwall *= -1;
	if (leftwall < 0) leftwall *= -1;
	Ogre::Vector3 normalvector = Ogre::Vector3(0, 1, 0);
	smallestvalue = floor;
	if (ceiling < smallestvalue){
		smallestvalue = ceiling;
		normalvector = Ogre::Vector3(0, -1, 0);
	}
	if (rightwall < smallestvalue){
		smallestvalue = rightwall;
		normalvector = Ogre::Vector3(1, 0, 0);
	}
	if (leftwall < smallestvalue){
		normalvector = Ogre::Vector3(-1, 0, 0);
	}
	return normalvector;
}
```

Refactoring 8 points

I refactored a lot of code during the project but the biggest two refactors were the collision and the movementphysics refactoring so I will be going over them.

Collision

The collision code before I refactored it looked like this

```
if (_gravityIsActive && otherdown)
	{
		//Ogre::LogManager::getSingleton().logMessage("otherdown: " +
Ogre::StringConverter::toString(otherdown));
		this->position.y -= velocity.y;
		this->velocity.y = 0;
	} else if (_gravityIsActive && other->getName() != "Player") {
		this->position = previousPos;
		this->velocity *= -0.2;
		this->velocity += other->velocity * 0.3;
		this->position += velocity;
	}
```

This function would get called if collision was detected. ‘otherdown’ was a hard-coded boolean that checked if the player was stading on another object, if this was the case than we only changed the vertical velocity and position. if that was not the case we did some wet finger calculation that looked semi acceptable and that was it. We decided that this was obviously not really an acceptable way of handing collision so I refactored it. After refactoring it looked like this.

```
if (!this->obj->isAffectedByPhysics() && other->obj->isAffectedByPhysics()){
	normalvector = getNormalVector(other, this);
	Ogre::Real dotproduct = normalvector.dotProduct(other->obj->velocity -
this->obj->velocity);
}
else if (this->obj->isAffectedByPhysics() && !other->obj->isAffectedByPhysics()){
	Ogre::Real dotproduct = normalvector.dotProduct(this->obj->velocity -
other->obj->velocity);
	this->obj->velocity += normalvector + (-normalvector * dotproduct);
}
else if (this->obj->isAffectedByPhysics() && other->obj->isAffectedByPhysics()){
	float number1, number2, closingvelocity;
	number1 = this->obj->velocity.dotProduct((other->obj->position -
this->obj->position).normalisedCopy());
	number2 = other->obj->velocity.dotProduct((this->obj->position -
other->obj->position).normalisedCopy());
	closingvelocity = number1 + number2;
	this->obj->velocity -= (other->obj->position -
this->obj->position).normalisedCopy() * closingvelocity;
	other->obj->velocity -= (this->obj->position -
other->obj->position).normalisedCopy() * closingvelocity;
}
```

This handles the collision in a mathematical correct way. It first checks wheter it is colliding with a static prop or a moving object. If it is colliding with a static prop it gets the normal vector and uses that to calculate the right change in velocity. If it is colliding with another moving object it first calculates the velocity both objects are supposed to have when done colliding and than calculates the direction vector for both objects.

Movement

We used to have one Movement vector that we would change whenever we needed to make changes to movement, so the inputmanager, collision and other code would all change this vector resulting in some unwanted behaviour. examples:
Collision code

```
this->obj->velocity += normalvector + (-normalvector * dotproduct);
```

InputManager code

```
//When the "W" key is pressed, add 6 to the velocity of y so the player moves up.
	case OIS::KC_W:
		_player->velocity.y = 7.5;
		break;
//When the "A" key is pressed, add -3 to the velocity of x so the player moves to the
left.
	case OIS::KC_A:
		_player->velocity.x = -2.5;
		break;
//When the "D" key is pressed, add 3 to the velocity of x so the player moves to the
right.
	case OIS::KC_D:
		_player->velocity.x = 2.5;
		break;
```

We decided to refactor this so we have multiple movement vectors that all combine into the movement vector

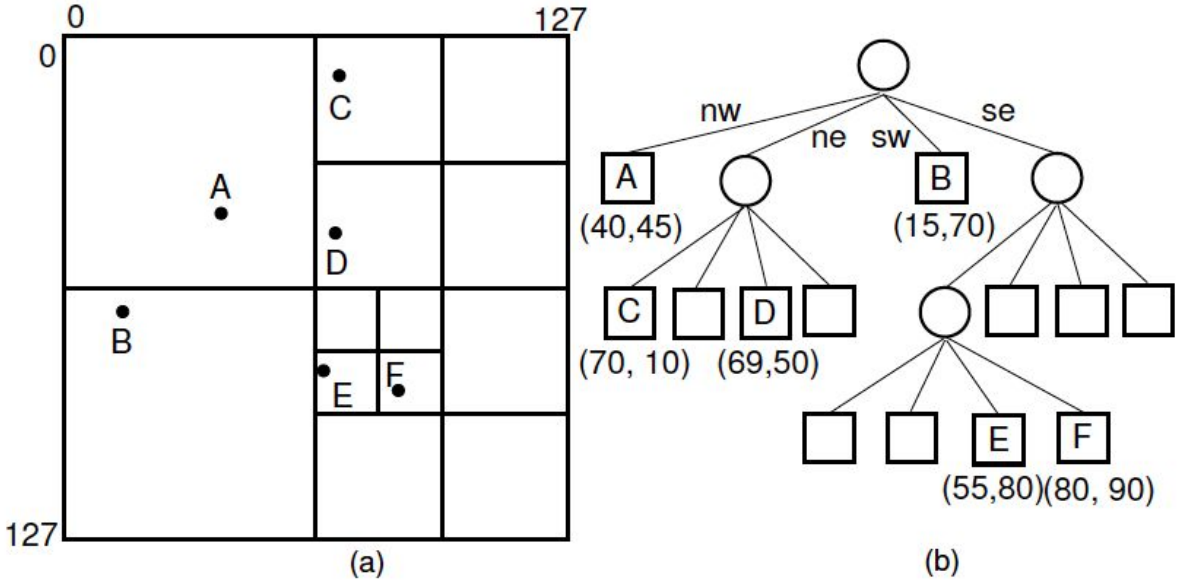
```
// Velocity
Ogre::Vector3 movementVelocity;
float maxMovementMagnitude;
Ogre::Vector3 gravityVelocity;
Ogre::Vector3 impulseVelocity;

Ogre::Vector3 getCombinedVelocity()
{
	return movementVelocity + gravityVelocity + impulseVelocity;
};
```

This way every part of the code can control a different part of the velocity leaving the other parts in tact.

Advanced techniques 6 points (x2)

I used a quad tree as advanced technique. We use this quadtree to detect collision. The quadtree works by dividing the field in quads. We based our quadtree on a Point-region quadtree[1]. In a Point quadtree each quad devides into four more quads as long as there are two or more objects left in the current quad. This results in a lot of quad that have either one or zero datapoints in it and ends up looking like this



However for this to be a good data structure for our game we do something a bit different. Due to our collision working with bounding boxes if a boundingbox fits in more than one quad we put in all of them. Than when the quads get to a certain minimum size and there is still more than 1 object in the quad we check for collision as we do here.

```
if (objectArray.size() > 1){
                /* comments here */
                if (getArea() < 40000){
                        auto it = objectArray.begin();
                        while (it != objectArray.end())
                        {
                                auto it2 = it + 1;
                                while (it2 != objectArray.end()){

(*it)->boundingbox->isCollidingWithAABB((*it2)->boundingbox);
                                        ++it2;
                                }
                                ++it;
                        }
                        return;
                }
                ...
}
```

The reason we do this is simple. If we only put the position in each quad there is a big chance some part of the boundingbox will overextend into the next quad and ignore collision over there. Doing this allows to check for way fewer collisions and thus speeding up the time needed for each update loop.

Conclusion

I really enjoyed the project and its technical challenges, at the end of the day it seems like I barely pass the requirements

Development basics (Knightstour)	10 points
C++	9 points
Tooling	8 points
coding standards	6 points
testing	2 points
porting	0 points
documentation	5 points
refactoring	8 points
advanced techniques	6 points (x2)
Total	60 points

So there you have it, I think I deserve a 6.0 grade for my Methodes and techniques. When I reflect on this project, I think I learned a lot, this grading system made me see where my strong and weak were in programming. I do have to note that I think it is kind of weird that pointing is part of the grading. I don't think there was any time for us to do that. I really enjoyed the project and look forward to the assesment.