设计模式课程设计

使用了SFML图形库，用了4周的时间搞了这个打飞机的游戏，考虑到作为一个学习性质的作业，从构思到编码采取了写框架的想法，在简易的游戏逻辑之外完成了简单地游戏引擎功能，并且尝试使用了C++11几乎所有的特性

采取了设计模式进行设计的部分如下

1）GUI部分（组件模式）

对应菜单的GUI功能需要。支持 文本标签/按钮/标签按钮组 三种组件

2）消息队列（命令模式）

游戏世界的抽象，将外部按键，游戏内发生的事件包装成统一的消息形式在世界内穿行

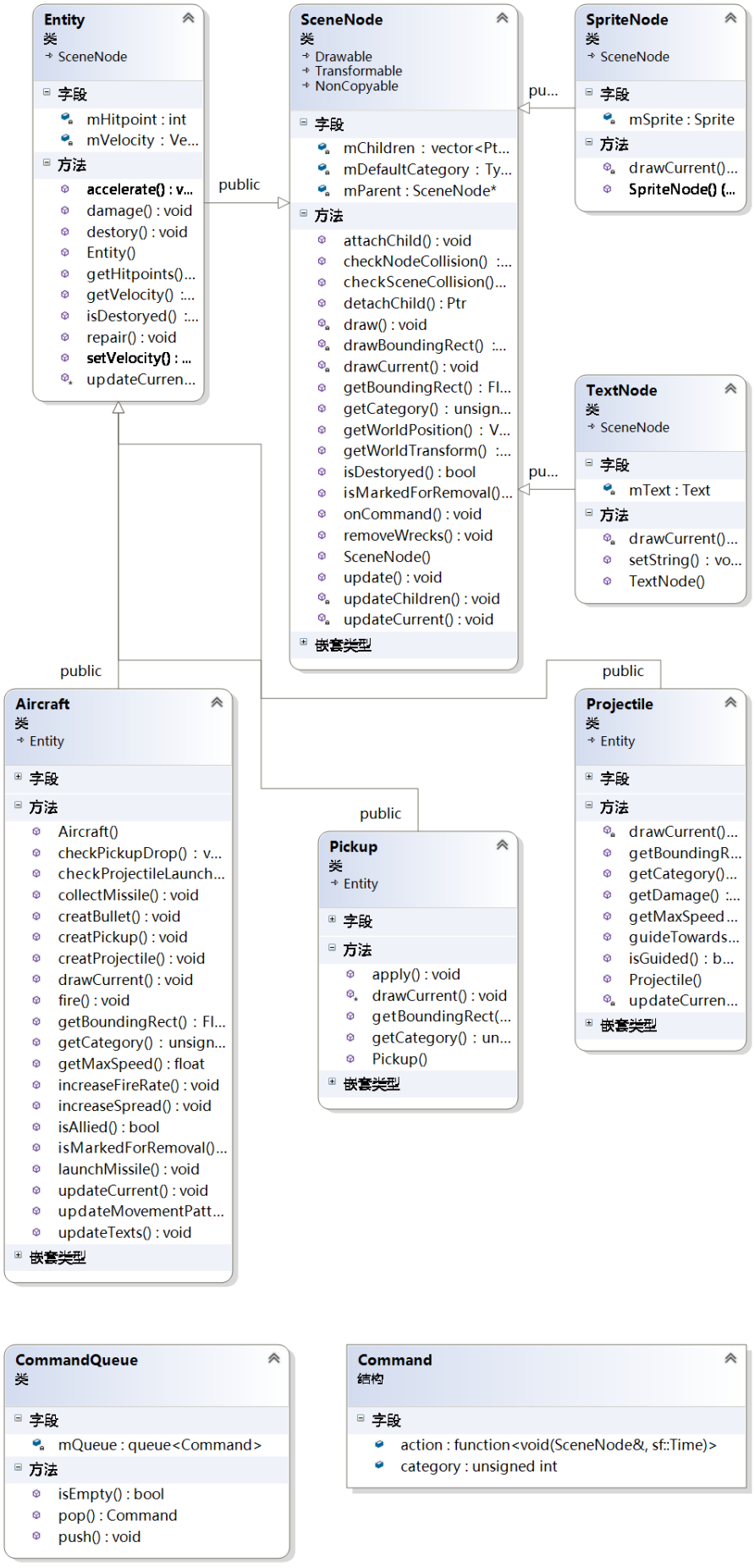
3）场景切换（外观模式）

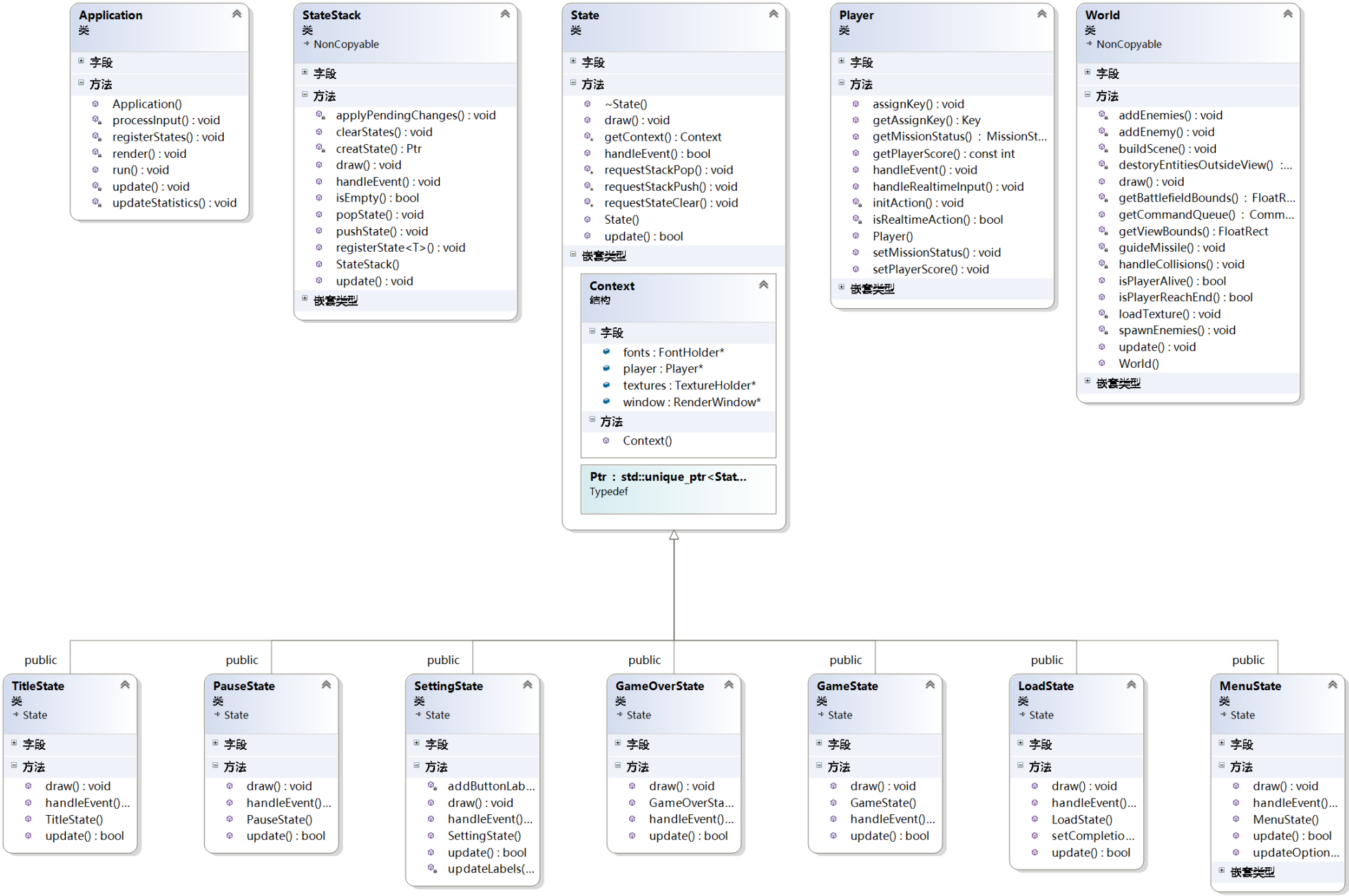
将游戏的不同界面（载入/主界面/暂停/菜单）进行抽象，支持多个场景叠加/同步更新

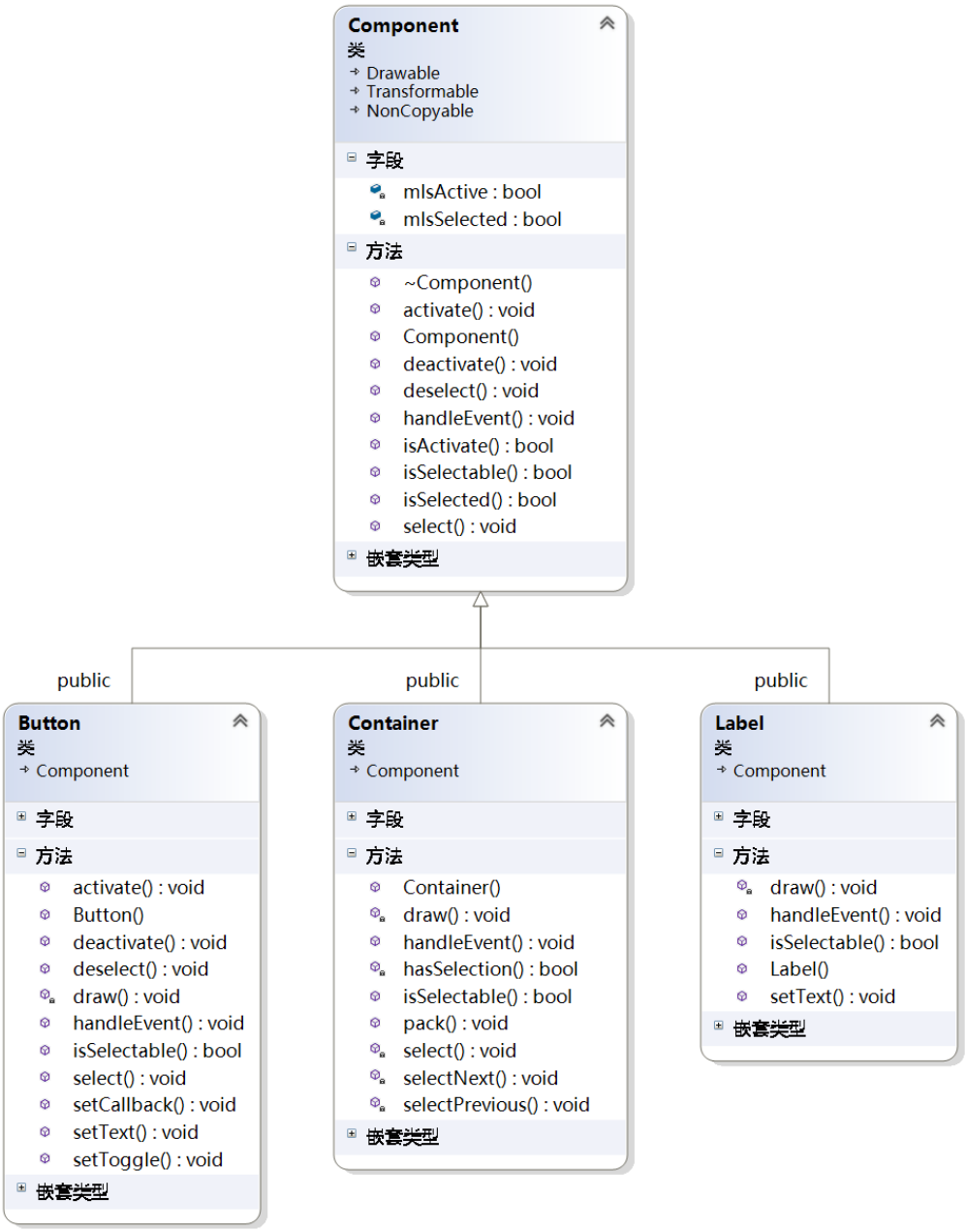
4）纹理节点树（组件模式）

将图层-精灵概念抽象。每一个图层都是一棵树，子节点与父节点之间的关系为相对位置，每次重绘按图层顺序使用DFS进行。

Release版本在VS2015+Win10下编译，UML图由VS的类图功能生成，比较简陋。目前版本引擎部分和游戏部分未完全分离，之后会进行迭代完善


Entity
类
SceneNode
字段
mHitpoint : int
mVelocity : Ve...
方法
accelerate() : v...
damage() : void
destory() : void
Entity()
getHitpoints()...
getVelocity() :...
isDestoryed() :...
repair() : void
setVelocity() : ...
updateCurren...
public
SceneNode
类
Drawable
Transformable
NonCopyable
字段
mChildren : vector<Pt...
mDefaultCategory : Ty...
mParent : SceneNode*
方法
attachChild() : void
checkNodeCollision() :...
checkSceneCollision()...
detachChild() : Ptr
draw() : void
drawBoundingRect() :...
drawCurrent() : void
getBoundingRect() : Fl...
getCategory() : unsign...
getWorldPosition() : V...
getWorldTransform() :...
isDestoryed() : bool
isMarkedForRemoval()...
onCommand() : void
removeWrecks() : void
SceneNode()
update() : void
updateChildren() : void
updateCurrent() : void
嵌套类型
pu...
SpriteNode
类
SceneNode
字段
mSprite : Sprite
方法
drawCurrent()...
SpriteNode() (...
pu...
TextNode
类
SceneNode
字段
mText : Text
方法
drawCurrent()...
setString() : vo...
TextNode()
public
public
Aircraft
类
Entity
字段
方法
Aircraft()
checkPickupDrop() : v...
checkProjectileLaunch...
collectMissile() : void
creatBullet() : void
creatPickup() : void
creatProjectile() : void
drawCurrent() : void
fire() : void
getBoundingRect() : Fl...
getCategory() : unsign...
getMaxSpeed() : float
increaseFireRate() : void
increaseSpread() : void
isAllied() : bool
isMarkedForRemoval()...
launchMissile() : void
updateCurrent() : void
updateMovementPatt...
updateTexts() : void
嵌套类型
public
Pickup
类
Entity
字段
方法
apply() : void
drawCurrent() : void
getBoundingRect(...
getCategory() : un...
Pickup()
嵌套类型
public
Projectile
类
Entity
字段
方法
drawCurrent()...
getBoundingR...
getCategory()...
getDamage() :...
getMaxSpeed...
guideTowards...
isGuided() : b...
Projectile()
updateCurren...
嵌套类型
CommandQueue
类
字段
mQueue : queue<Command>
方法
isEmpty() : bool
pop() : Command
push() : void
Command
结构
字段
action : function<void(SceneNode&, sf::Time)>
category : unsigned int


Application
类
字段
方法
Application()
processInput() : void
registerStates() : void
render() : void
run() : void
update() : void
updateStatistics() : void
StateStack
类
NonCopyable
字段
方法
applyPendingChanges() : void
clearStates() : void
creatState() : Ptr
draw() : void
handleEvent() : void
isEmpty() : bool
popState() : void
pushState() : void
registerState<T>() : void
StateStack()
update() : void
嵌套类型
State
类
字段
方法
~State()
draw() : void
getContext() : Context
handleEvent() : bool
requestStackPop() : void
requestStackPush() : void
requestStateClear() : void
State()
update() : bool
嵌套类型
Context
结构
字段
fonts : FontHolder*
player : Player*
textures : TextureHolder*
window : RenderWindow*
方法
Context()
Ptr : std::unique_ptr<Stat...
Typedef
Player
类
字段
方法
assignKey() : void
getAssignKey() : Key
getMissionStatus() : MissionSt...
getPlayerScore() : const int
handleEvent() : void
handleRealtimeInput() : void
initAction() : void
isRealtimeAction() : bool
Player()
setMissionStatus() : void
setPlayerScore() : void
嵌套类型
World
类
NonCopyable
字段
方法
addEnemies() : void
addEnemy() : void
buildScene() : void
destoryEntitiesOutsideView() :...
draw() : void
getBattlefieldBounds() : FloatR...
getCommandQueue() : Comm...
getViewBounds() : FloatRect
guideMissile() : void
handleCollisions() : void
isPlayerAlive() : bool
isPlayerReachEnd() : bool
loadTexture() : void
spawnEnemies() : void
update() : void
World()
嵌套类型
public
TitleState
类
State
字段
方法
draw() : void
handleEvent()...
TitleState()
update() : bool
public
PauseState
类
State
字段
方法
draw() : void
handleEvent()...
PauseState()
update() : bool
public
SettingState
类
State
字段
方法
addButtonLab...
draw() : void
handleEvent()...
SettingState()
update() : bool
updateLabels(...
public
GameOverState
类
State
字段
方法
draw() : void
GameOverSta...
handleEvent()...
update() : bool
public
GameState
类
State
字段
方法
draw() : void
GameState()
handleEvent()...
update() : bool
public
LoadState
类
State
字段
方法
draw() : void
handleEvent()...
LoadState()
setCompletio...
update() : bool
public
MenuState
类
State
字段
方法
draw() : void
handleEvent()...
MenuState()
update() : bool
updateOption...
嵌套类型


Component
类
Drawable
Transformable
NonCopyable
字段
mIsActive : bool
mIsSelected : bool
方法
~Component()
activate() : void
Component()
deactivate() : void
deselect() : void
handleEvent() : void
isActivate() : bool
isSelectable() : bool
isSelected() : bool
select() : void
嵌套类型
public
public
public
Button
类
Component
字段
方法
activate() : void
Button()
deactivate() : void
deselect() : void
draw() : void
handleEvent() : void
isSelectable() : bool
select() : void
setCallback() : void
setText() : void
setToggle() : void
嵌套类型
Container
类
Component
字段
方法
Container()
draw() : void
handleEvent() : void
hasSelection() : bool
isSelectable() : bool
pack() : void
select() : void
selectNext() : void
selectPrevious() : void
嵌套类型
Label
类
Component
字段
方法
draw() : void
handleEvent() : void
isSelectable() : bool
Label()
setText() : void
嵌套类型

外观模式：

```
class State {
public:
	typedef std::unique_ptr<State> Ptr;
	struct Context {
		Context(sf::RenderWindow& window,TextureHolder& textures,FontHolder&
fonts,Player& player);

		sf::RenderWindow* window;
		TextureHolder* textures;
		FontHolder* fonts;
		Player* player;
	};

public:
	State(StateStack & stack, Context context);
	virtual ~State();
	virtual void draw() = 0;
	virtual bool update(sf::Time dt) = 0;
	virtual bool handleEvent(const sf::Event &event) = 0;

protected:
	void requestStackPush(States::ID stateID);
	void requestStackPop();
	void requestStateClear();

	Context getContext() const;

private:
	StateStack* mStack;
	Context mContext;
};
```

运用在对整个世界的抽象——所有界面的变动由一个deltaT驱动，用于控制的类只需要对某个世界发出update(dt)即可完成

State为一个虚基类。游戏/暂停/换场/菜单/设置均为State的继承，由一个栈来维护。delta的发出从栈顶向下进行，重绘采取相同的顺序

组件模式：

```
class SceneNode : public sf::Drawable, public sf::Transformable,private sf::NonCopyable{
public:
	typedef std::unique_ptr<SceneNode> Ptr;
	typedef std::pair<SceneNode*, SceneNode*> Pair;
public:
	explicit SceneNode(Category::Type type = Category::None);
	void attachChild(Ptr child);
	Ptr detachChild(const SceneNode& node);

	void update(sf::Time dt,CommandQueue &commands);

	virtual unsigned int getCategory() const;
	void onCommand(const Command &command, sf::Time dt);

private:
	virtual void updateCurrent(sf::Time dt,CommandQueue &commands);
	void updateChildren(sf::Time dt,CommandQueue &commands);

	virtual void draw(sf::RenderTarget &target, sf::RenderStates status)const;
	virtual void drawCurrent(sf::RenderTarget &target, sf::RenderStates status)const;
	void drawBoundingRect(sf::RenderTarget& target, sf::RenderStates) const;

private:
	std::vector<Ptr> mChildren;
	SceneNode* mParent;
	Category::Type mDefaultCategory;
};

void SceneNode::update(sf::Time dt,CommandQueue &commands)
{
	updateCurrent(dt,commands);
	updateChildren(dt,commands);
}
void SceneNode::updateChildren(sf::Time dt,CommandQueue &commands)
{
	for (auto it = mChildren.begin(); it != mChildren.end(); ++it) {
		(*it)->update(dt,commands);
	}
}
```

采用了组件的设计模式将原本离散的屏幕上的各个部分按照一定的逻辑组合在一起。例如实现了图层（背景层，飞机层，特效遮罩层）等。相关信息可以以相对位置表示（飞机的血条，导弹数量）。相对关系可以在树结构的DFS过程中的得到。整个update与draw的过程都是使用DFS的方式实现，消息的传递同样。若移除游戏内某个对象（例如飞机爆炸被清除），则直接递归删除掉这个树节点以及所有儿子即可

命令模式：

```
struct Command {
	std::function<void(SceneNode&, sf::Time)> action;
	unsigned int category;
};
class CommandQueue {
public:
	void push(const Command &command);
	Command pop();
	bool isEmpty() const;
private:
	std::queue<Command> mQueue;
};
void SceneNode::onCommand(const Command & command, sf::Time dt)
{
	if (command.category & getCategory()) {
		command.action(*this, dt);
	}
	for (auto it = mChildren.begin(); it != mChildren.end(); ++it) {
		(*it)->onCommand(command, dt);
	}
}
```

对这个游戏世界的操作采用了命令模式来控制。使用了一个消息队列来实现。每个消息有对应的相应类型与消息事件。消息事件是一个将本节点当做参数的functional类型，这个设计可以在命令内完成复杂的逻辑过程。所有飞机的移动，子弹/炮弹的发射，子弹追踪位置计算都由消息来控制。
在实现的过程中，不同的消息在不同的地方定义，可以是不同的类型，但是形参必须相同，这样可以用C++11的functional进行封装，大大增加了灵活性。