Что сделано?

Язык программирования – C++, Qt 5.6.0 MinGW 32bit

Среда разработки – Qt Creator последней версии

Накидана модель движка, общий концепт которой можно представить как «Событие запускает триггер»

Каждое внутриигровое событие, будь то перемещение, добавление или удаление объекта на поле, действия игроков, смена статусов, описано в отдельном классе, унаследованном от абстрактного класса EventObject. Все типы событий перечислены отдельным enum-ом EVENT\_TYPE, и для каждого типа подразумевается реализация отдельного подкласса.

В каждом отдельном классе события прописаны т.н. триггеры, или реакции движка игры на события. Триггеры описываются семейством классов, унаследованных от абстрактного класса Trigger, аналогично с классами событий. Само понятие триггера пришлось отделить от понятия события по следующим причинам:

А) каждое событие может запускать своим срабатыванием целую цепочку триггеров

Б) игра подразумевает постоянную смену каких-то отдельных правил (например, изменение игровых параметров в зависимости от конкретного Древнего, или ввод в игру Процессов и Слухов). Эти правки можно так же представить в виде отдельных триггеров, висящих в отдельном пуле и срабатывающих при запуске конкретного события и соблюдении дополнительных условий.

В) Триггерами можно полностью описать любую реакцию движка, как игромеханическую, так и графическую

Оба понятия – «Триггер» и «Событие» - полностью взаимосвязаны. Любое событие подразумевает запуск определенных триггеров, в любом триггере можно прописать порождение и запуск определенных событий.

Пример – что и как происходит при выполнении события «Разыгрывается карта Мифа»

Класс Mythos, который описывает события, разыгрываемые во время фазы Мифа – по сути, все, что происходит при разыгрывании карты Мифа в реальной игре

class Mythos: public IObject

{

private:

QList<EventObject\*> eventResolveList

public:

Mythos(QString name);

void appendEventToResolveList(EventObject\* event);

void ResolveEvents();

};

eventResolveList – список, содержащий конкретные события, выполняемые в порядке добавления методом appendEventToResolveList(). Метод ResolveEvents()вызывается для начала выполнений события из триггера TriggerResolveCard – по сути, единственное действие, выполняемое этим триггером.

void Mythos::ResolveEvents()

{

foreach (EventObject\* event, eventResolveList) {

event->runEvent();

}

}

Сам класс унаследован от IObject, как и любой внутриигровой объект, для которого можно описать его местоположение относительно поля или других объектов

Вот так может выглядеть инициирование конкретного Мифа событиями:

Mythos\* myth = new Mythos("In other Space");

myth-> appendEventToResolveList(EventObject::createEvent(TT\_GATE\_BURST,Data::getRandomFreeGate(),Data::getLocationByName("Woods")));

myth-> appendEventToResolveList(EventObject::createEvent(TT\_SYMBOL\_MOVE,MARK\_MOON,MARK\_WHITE));

myth-> appendEventToResolveList(EventObject::createEvent(TT\_SYMBOL\_MOVE,MARK\_PLUS,MARK\_BLACK));

insertGlobalMythosDeck(myth);

Первое событие – прорыв врат, второе и третье – движение монстров с определенными символами по определенным стрелкам. Метод insertGlobalMythosDeck() добавляет Миф в «колоду Мифа»

Теперь эта карта может быть разыграна в любой момент при выполнении следующей строки:

EventObject::createEvent (TT\_RESOLVE\_CARD, Data::getData() -> getUpperMythos(), CARD\_MYTHOS) -> runEvent();

EventObject::createEvent() – статический метод, который создает (а точнее, копирует уже созданный, т.к. классы событий и триггеров построены на основе фабричного метода) объект EventResolveCard, получив первым параметром тип события TT\_RESOLVE\_CARD, вторым – указатель на карту Мифа (точнее, на объект IObject), полученную с помощью getUpperMythos(), и третьим – тип разыгрываемой карты, CARD\_MYTHOS. Поскольку хранить созданное событие нам не нужно, сразу же вызывается метод runEvent(), описанный в EventObject

void EventObject::runEvent()

{

*iniEventTrigger*();

bool interrupt = false; //прерывание после выполнения пула триггеров

foreach (Trigger \*trigger, Data::getData()->triggerPool) { //проверка пула триггеров для выполнения подходящих тригеров ДО события

if((this->what == trigger->condition) && trigger->checkTriggerConditions()){

interrupt |= trigger->applyTrigger(this);

}

}

if(!interrupt) //если прерывания не было, применить действия

foreach (Trigger \*trigger, triggerList) {

if(trigger->checkTriggerConditions())

trigger->*applyTrigger*(); //выполнение триггеров события

}

//TODO обеспечить удаление события после запуска

}

В виртуальном методе *iniEventTrigger*() происходит инициирование триггеров, связанных с событием. Делается это в начале выполнения события, чтобы не вводить механизм обновления событий, если они ждут своей очереди на выполнение, а игровая ситуация успела измениться. Вроде бы так =) Для EventResolveCard этот метод определен следующим образом:

void EventResolveCard::*iniEventTrigger*() //тип карты записан в commTarget

{

appendTrigger(Trigger::createTrigger(TT\_RESOLVE\_CARD,who,target));

}

Автоматизировать добавление триггеров по типу не получится, потому что для каждого типа события набор триггеров, и даже их количество, могут варьироваться. Все приходится прописывать вручную.

Дальше идет просмотр пула триггеров, и те, что имеют такой же тип в condition, что и тип события, запускаются на выполнение методом applyTrigger(). Переменная bool interrupt введена для случаев, когда выполнение триггеров из пула отменяет запуск триггеров самого события (игровые ситуации типа «Когда А, вместо Б сделайте В»). Если после просмотра пула переменная продолжает быть false, то запускается собственный список триггеров события.

Метод *applyTrigger()* для TriggerResolveCard определен ниже

void TriggerResolveCard::*applyTrigger*()

{

qDebug()<<"Trigger: resolve card";

static\_cast<Mythos\*>(object)->ResolveEvents();

}

Что конкретно реализовано на данный момент?

Абстрактные классы EventObject и Trigger, подклассы для них (21 из примерно 45 различных типов), механизм инициирования и выполнения событий и триггеров.

Перечисления: EVENT\_TYPE, PHASE, ABILITY, ACTION, OBJECT\_TYPE, MONSTER, MONSTER\_SPECIAL, CARD, ITEM\_TYPE, ITEM\_SETTING, EFFECT, SETTING

Для большинства из них даны комментарии

IObject самый низкоуровневый базовый класс для большинства игровых объектов

class IObject

{

protected:

OBJECT\_TYPE objectType;

int ID;

public:

QList<IObject\*> tokenPool; //список объектов "сверху" на объекте

IObject\* place; //местонахождение объекта (локация или сыщик или другой объект)

QString name;

IObject(QString name, OBJECT\_TYPE type);

OBJECT\_TYPE getObjectType() const;

void setObjectType(const OBJECT\_TYPE &value);

int getIObjectID();

void appendToPlace(IObject\* object);

void removeFromPlace();

};

Token, унаследован от него, описывает объекты, которые могут быть физически помещены на поле

class Token:public IObject

{

public:

Token(QString name, OBJECT\_TYPE type);

void moveToken(IObject \*to); //переместить жетон по полю

void placeToken(IObject\* to); //поместить жетон на поле

void removeToken(); //снять жетон с поля

};

От него унаследованы Monster и Gate. Для монстров определен виртуальный метод *MonsterMove(),* и созданы подклассы типа MonsterBlack , которые описывают разных монстров по цвету их рамки

Описан класс Card, наследуемый от IObject

protected:

bool isExhausted;

short useHand;

short primeParam; //первый параметр на карте; +X к проверкам, и т.д.

CARD cardType;

ITEM\_TYPE itemType;

QList <ITEM\_SETTING> itemSettings;

public:

Card(QString name, CARD card\_type, OBJECT\_TYPE type);

virtual void *appendToPlayer*(IObject\* player)

{

appendToPlace(player);

} //пассивные свойства - триггеры

virtual void *removeFromPlayer*() //удаление пассивных свойств

{

*stopUseCardAction*();

removeFromPlace();

}

virtual void *useCardAction*(){} //задействование карты

virtual void *stopUseCardAction*(){}

short getUseHand() const;

CARD getCardType() const;

};

Для каждой колоды предметов выделяется отдельный класс

class CardCommonItem:public Card

{

protected:

short price;

public:

CardCommonItem(QString name);

};

Дальше либо для каждого предмета создается отдельный подкласс (удобно прописывать свойства и триггеры для всех виртуальных методов), либо создать огромный конструктор или фабрику, которой можно передавать названия предметов и получать на выходе сами предметы – еще не решено

Описан класс локации – отдельной зоны на игровом поле

class location: public IObject

{

int id;

int idDistrict; //номер района

bool isStable; //true - зеленый ромб, false - красный

bool isStreet; //true - улица, false - локация

public:

location(QString name, int idDistrict, bool isStable, bool isStreet);

location\* Black; //указатель на смежную локацию по белой стрелке

location\* White; //указатель на смежную локацию по черной стрелке

QList <location\*> LstCommonPath; //список смежных локаций для сыщиков

QList <Monster\*> getMonsters(); //возвращает список всех находящихся в локации монстров

Gate\* getGate(); //возвращает указатель на жетон врат, открытых в локации, или 0, если там их нет

bool containPlayer(); //true если содержит хотя бы одного игрока, false

bool containAquatic(); //true содержит жетон водной локации, false - нет

Token\* getSeal();

QList <investigator\*> getPlayersOnLocation();

bool getIsStreet() const;

void setIsStreet(bool value);

};

Наконец описан класс игрока, который предполагается наследовать конкретным игрокам

class investigator:public Token

{

protected:

int id;

QMap<EFFECT, int> status; //эффект - значение

short currStamina;

short StaminaMaximum;

short currSanity;

short SanityMaximum;

short Focus;

short currFocus;

short freeHands;

short currHands;

short money;

QVector<short>SneakTrack;

QVector<short>SpeedTrack;

QVector<short>FightTrack;

QVector<short>WillTrack;

QVector<short>LoreTrack;

QVector<short>LuckTrack;

short posSneakSpeed; //позиция ползунка на первой полоске 0-3

short posFightWill; //позиция ползунка на второй полоске 0-3

short posLoreLuck; //позиция ползунка на третьей полоске 0-3

short currSpeed;

short currSneak;

short currFight;

short currWill;

short currLore;

short currLuck;

short modifyCombat;

short modifyHorror;

short modifySpell;

short modifyEvade;

QList <Card\*> inventoryCard;

QList <Card\*> cardInHands; //для оружия в бою

QList <Token\*> personalClueCup;

QList <Token\*> personalPowerCup;

QList <Token\*> personalMonsterTrophy;

QList <Token\*> personalGateTrophy;

public:

virtual void *increaseSneak*();

virtual void *increaseSpeed*();

virtual void *increaseWill*();

virtual void *increaseFight*();

virtual void *increaseLore*();

virtual void *increaseLuck*();

investigator(QString playername);

int RollDice(int count, int pen);

int OneRollDie(int N);

int CheckAbility(ABILITY check, int pen, int limit);

short getSneak() const;

void setSneak(short value);

short getModifyCombat() const;

void setModifyCombat(short value);

short getModifyHorror() const;

void setModifyHorror(short value);

short getModifySpell() const;

void setModifySpell(short value);

short getModifyEvade() const;

void setModifyEvade(short value);

short getCurrHands() const;

void setCurrHands(short value);

short getFreeHands() const;

void setFreeHands(short value);

void appendCardInHands(Card\* weapon);

void removeCardFromHands(Card\* weapon);

void appendObjectToInventory(IObject\* object);

void removeObjectFromInventory(IObject\* object);

};

ЧТО ДЕЛАТЬ ДАЛЬШЕ?

Первый этап:

* Дописывание механизма движка. Реализация всех событий и триггеров, насколько это возможно. Реализация условных триггеров (работающих в пуле триггеров), либо внедрением допкласса условий, либо альтернативой.
* Реализация простой графической модели – поле с тремя-четырьмя локациями, окраинами, небом и одним иным миром. По полю можно перетаскивать фишки, причем для каждой четко фиксируется ее текущее положение и места куда ее можно поместить. Прикрутить механизм событий-триггеров к графической модели – фишки должны появляться и пропадать с поля, посылая соответствующие события или согласно триггерам. Рядом – место для листа персонажа, с возможностью менять статы, задействовать и сбрасывать предметы с инвентаря.
* Реализация редактора предметов – сборника форм, которая содержит кучу полей со значениями, которых достаточно для полного описания любого предмета в игре, карточки или жетона. Выбрать протокол серилизации, серилизовывать структуру предмета в файл, сделать чтение из файла в игру, после которого база данных заполнится игровыми объектами.
* Тестирование переносимости на Android – для эмулятора и для реального устройства. Написать простое приложение с QOpenGL или QGraphics, взятое из любого примера, и успешно его запустить на удаленном устройстве.

Второй этап:

* Разработка и разметка игрового интерфейса. Разбивка на отдельные игровые экраны (Главное меню, лобби, выбор персонажа, настройки игры, сама игра). Расположение панелей, баров, карты, счетчиков, вызов инвентаря.
* Написание модели сетевой игры – как будет осуществляться поиск игроков, подключение, игровой процесс. Нужно ли выделять отдельный сервер, или каждый клиент будет включать сервер в себе. Написание концепта базы данных для сетевой игры.
* Выбор средства разработки графического движка. Варианты: QML (личный минус), QGraphics (нейтрально – разобрался, как с ним работать, но довольно громоздко), QOpenGL – (не разбирался, как работать, но он мощнее и оптимизированнее, чем QGraphics). Либо что-то стороннее и свободно распространяемое, типа Box2D или Unity.
* Наполнение. Поиск изображений-спрайтов – портретов игроков, предметов, поля, монстров, подгон в максимально возможное качество и к единому стандарту. Поиск звукового и музыкального контента.
* Заполнение игровой базы объектов с помощью написанного ранее редактора предметов – несложно, но довольно кропотливо
* Диздок, включающий в себя указания по интерфейсу, игровому процессу, задействованным областям.

Третий этап:

* Написание графического движка с помощью выбранной технологии и с учетом изготовленной на первом этапе модели. Привязка механики событий-триггеров, как в разработанной модели.
* Написание игровой механики – начиная с выбора персонажа и весь игровой процесс далее, используя механизм событий-триггеров.
* Написание базы данных и механизма сетевой игры.
* Сбор и отладка отдельно PC и Android версий.

Четвертый этап:

* Тестирование
* Тестирование
* Тестирование

Пятый этап (опционально):

* Распространение
* Разговор с правообладателями, возможно выпуск и кучи денег в карманы, помимо кайфа от самой разработки и последующих игр