SQUARE ENIX

ver 2008.09.08

KAMIO TAKASHI

Squirrelを使ったゲーム開発

ごあいさつ

Hello squirrel world!

アジェンダ

- The programming language squirrel
  - 概要
  - 簡単な解説
- Wiiウェア小さな王様と約束の国
  ファイナルファンタジー・クリスタルクロニクル
  - システムの特徴や、良かった点、問題点など
- まとめ
- Q&A

The programming language squirrel

概要

概要

- Squirrel ってなんて読むの？
- Alberto Demichelis さん作。
- Squirrel はリスのこと。
- スクリプトの拡張子は .nut。
- 組み込みスクリプト。
- シンタックスが C/C++ライク。
- ソースコードつきである。
- 6000行ほどのＣ＋＋コードで書かれている。
- 組み込みが容易である。
- alloc, free などが簡単に置き換え可能になっている。
- zlib/libpng ライセンス 。

概要

- 型は動的型。
- ＧＣ（ガベージコレクタ）。
- クラスがあり、継承もできる。
- グローバル変数には初期化方法に制限がある。
- integerとfloatに区別がある。（Lua には無くて改良点の一つ）
- ビット演算のサポート。（これも改良点）
- Squrrel -> C++, C++ -> Squirrel 双方向で呼び出しや参照が可能
- SQDev - Eclipse を使ったデバッガがある。
- SqPlus という便利なバインダ。
- 設定を変更すれば、Unicode も扱える。
- コンパイルすればWindowsのコマンドラインでも動かせる。
- クロスコンパイルも可能。

The programming language squirrel

基本機能の説明

制御文

if

```
if( 100 < a ){
        // 処理
}
else
if( 10 < a ){
        // 処理
}
else{
        // 処理
}
```

while

```
while( i < 10 ){
        // 処理
        i++;
}
```

for

```
for( i=0; i<10; i++ ){
        // 処理
}
```

switch

```
switch( a ){
        case 0:
                // 処理
                break;
        case 1:
                // 処理
                break;
        default:
                // 処理
                break;
}
```

制御文

- if

```
if( 100 < a ){
        // 処理
}
else
i
}
e
}
```

- while

```
while( i < 10 ){
        // 処理
        i++;
}
```

- for

```
for( i=0; i<10; i++ ){
        // 処理
}
```

```
        case 0:
                // 処理
                break;
        case 1:
                // 処理
                break;
        default:
                // 処理
                break;
}
```

C/C++そのまま

関数

- 引数なしの関数の例

```
function hello()
{
    print("hello Squirrel!");
}
```

- 引数あり、戻り値ありの関数の例

```
function hello( a, b )
{
    return (a + b);
}
```

ローカル変数

- 初期化した時点で型が決まる

```
function hello ()
{
        local b = true;    // bool
        local a = 1;       // integer
        local f = 10.0;    // float
        local s = "foo";   // string
}
```

- 型変換

```
a.tofloat();       // 1 -> 1.0
f.tointeger();     // 10.0 -> 10
```

グローバル変数

- グローバル変数はいきなり代入は出来ない。

```
foo = 10; // ERROR
```

- 正しい初期化の仕方。

```
foo <- 10; // OK
//又は
::foo <- 10; // OK
```

- :: を使ってスコープを指定

```
function test()
{
    local foo = 123;

    print(“foo=”+ ::foo +“¥n”);  //> 10 グローバル
    print("foo="+ foo +"¥n");    //> 123
}
```

配列

- 初期化

```
local fooArray = [ 0, 1, 2, 3, 4, "five", 6.0 ];
```

- 初期化

```
local fooArray = array( 2 ); // サイズ2の配列 値は null
```

- 追加削除

```
fooArray.append( 100 );
fooArray.remove( 0 ); //idx
```

テーブル

- 初期化

```
local fooTable = {
    a = 10,
    b = 20,
    c = 30,
};
```

- 要素なしで初期化

```
local fooTable = {};
```

- 追加

```
fooTable.d <- 0; // .d が足され値がセットされる
```

- アクセス

```
fooTable["d"] = 30; //どちらもアクセス可能

fooTable.d = 40;    //どちらもアクセス可能
```

- キーの有無の判定

```
if( "foo" in fooTable ){
    // fooTable .foo が有る
}
else{
    // fooTable .foo は無い
}
```

foreach

- 配列

```
foreach( i, val in fooArray ){
    print(“fooArray[”+ i +”]=”+ val +"¥n")
}
```

```
fooArray[0]=10
fooArray[1]=20
fooArray[2]=30
```

- テーブル

```
foreach( key, val in fooTable ){
        print("fooTable."+ key +”=”+ val +"¥n")
}
```

```
fooTable.a=10
fooTable.b=20
fooTable.c=30
```

クラス

```
class Foo
{
    _fooTable = null;

    constructor()
    {
        _fooTable = {};
    }
}
```

- テーブルと同じ仕組を利用して作成されている。
- 派生が可能。
- C++バインドクラスからの派生も可能。
- コンストラクタはあるが、デストラクタはない。

クラスのstaticメンバー

```
class Foo
{
    static _staticValue = 10;

    static _staticArray =
    [
        10,
        20,
    ];

    static _staticTable =
    {
        a=10,
        b="str",
    };


    static function IsFoo()
    {
        return true;
    }
}
```

- static メンバーも使用できる。

- アクセス方法

```
Foo::_staticValue

Foo::_staticArray[0]

Foo::_staticTable.b

Foo::IsFoo()
```

typeof

- 動的に初期化された値の型は、typeof で型を調べることが出来る

```
local i = 0;
local f = 0.0;
local s = "";
local c = Foo(); //class Foo
local a = [];
local t = {};

print( typeof( i ) +"¥n");
print( typeof( f ) +"¥n");
print( typeof( s ) +"¥n");
print( typeof( Foo ) +"¥n");
print( typeof( a ) +"¥n");
print( typeof( t ) +"¥n");
print( typeof( c ) +"¥n");
```

```
integer
float
string
class
array
table
instance
```

- typeofでは、クラスのインスタンスは instance とだけ表示される

insntaceof

- instanceofを使うと insntace の型を調べることが出来る。

```
class Foo
{
}

class Bar extends Foo
{
}
```

```
local a = Bar(); //class Bar

if( a instanceof ::Hoge ){
    print("Hoge¥n");
}
if( a instanceof ::Bar ){
    print("Bar¥n");
}
if( a instanceof ::Foo ){
    print("Foo¥n");
}
```

```
Bar
Foo
```

- a は Bar のインスタンスであり、派生元が Foo なので、Bar と Foo がprint される

clone

- 代入すると参照が渡される。
- 値を明示的にコピーしたい場合は clone を使う。

- 参照先を変更してしまった例

```
local fooArray = [];

fooArray.append( 10 );

local barArray = fooArray;

barArray.append( 20 );
```

```
fooArray[0] = 10
fooArray[1] = 20
```

```
barArray[0] = 10
barArray[1] = 20
```

- clone で複製した例

```
local fooArray = [];

fooArray.append( 10 );

local barArray = clone( fooArray );

barArray.append( 20 );
```

```
fooArray[0] = 10
```

```
barArray[0] = 10
barArray[1] = 20
```

SqPlusを使ったバインド

- 関数のバインド

```
SqPlus::RegisterGlobal( Function, _T(“Function") );
```

  - 引数が７個より多いとエラーがでます。

- 定数のバインド

```
#define MODE_NORMAL      0
SqPlus::BindConstant( static_cast<int>(MODE_NORMAL), "MODE_NORMAL" );
```

  - enum や変数でも値ならバインドできます。

SqPlusを使ったバインド

- クラスのバインド

```
SqPlus::SQClassDef< Vector >( "Vector" )
    .var( &Vector::x, "x" )
    .var( &Vector::y, "y" )
    .var( &Vector::z, "z" )
    .func( &Vector ::SetX, “SetX”)
    .func( &Vector ::GetX, “GetX”);
```

  - namespaceなどを使っていても大丈夫。

URL

Squirrel

http://squirrel-lang.org/

SqPlus

http://wiki.squirrel-lang.org/default.aspx/SquirrelWiki/SqPlus.html

SQUARE ENIX

小さな王様はこんなゲーム

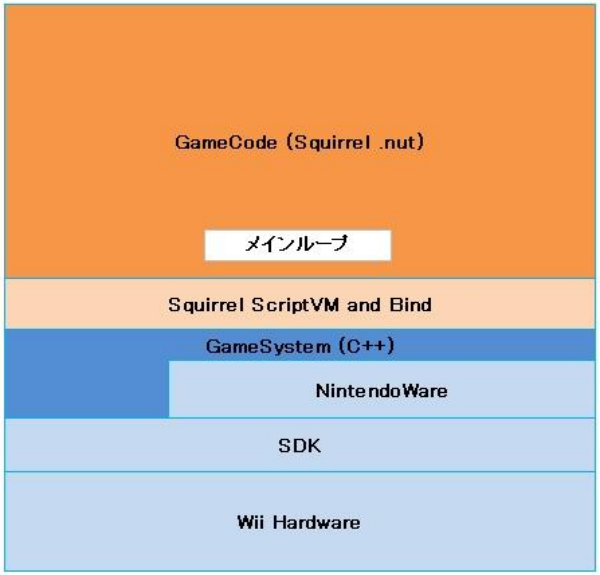
小さな王様はこんなゲーム

私たちがやろうとしたこと

- スクウェア・エニックスにとって新しい挑戦！
  - スモールな開発　短期間、少人数。
- 新しいゲームデザイン。
  - 仕様変更とトライアンドエラー。
- Wii ウェア。

これらを実現するために

特徴

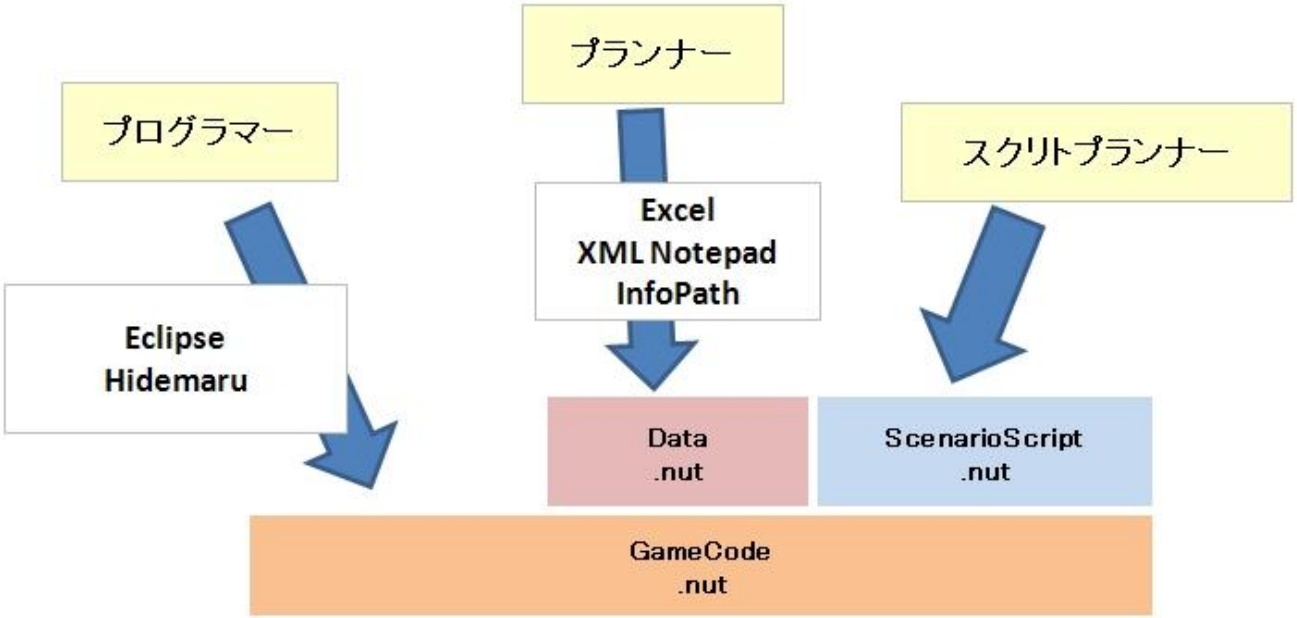


約7割

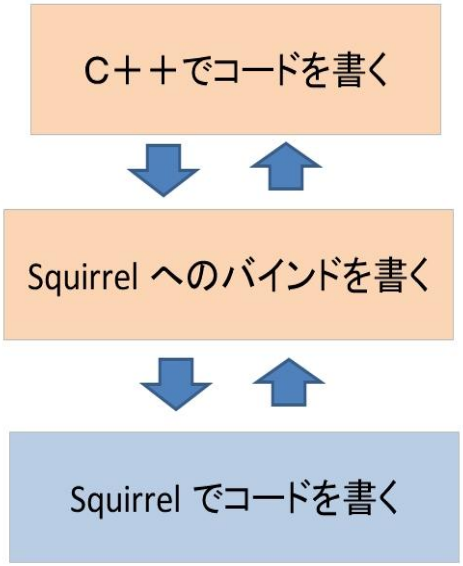
- SDK + NintendoWare それに Squirrel (SqPlus)
- 約７割のプログラムが Squirrel で書かれている。
- メインループも Squirrel 側にある。
- いわゆるシステム部分はC++、ゲーム部分が Squirrel。
- 純粋にスクリプトのオーバヘッドと呼べるのは
  CPUの10%程度で大半はその他の処理。
- SQDev はUSB通信速度がボトルネックになり、
  結果的に実用的なレベルでは使うことが出来なかった。

Squirrelを中心としたワークフロー

- 全ての箇所で Squirrel が使われています。



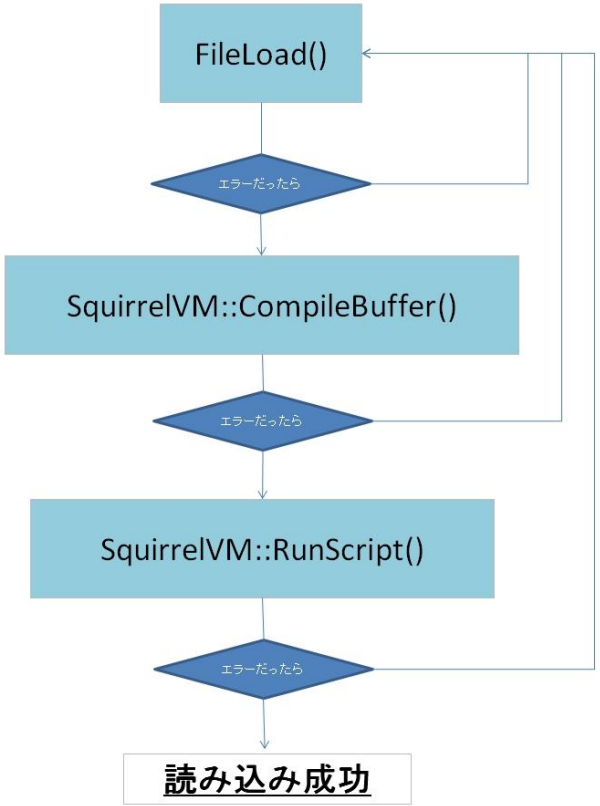
プログラマのワークフロー



- 序盤はC++コードをメインに作成。
- 中盤からC++とSquirrelを行ったりきたり。
- 終盤は必要に応じて、C++コードを加筆して Squirrel にバインド。

- 結構このバインドが面倒。
  - バインドをツールで半自動化して効率化。
- 動的型なので、何が入ってくる解り難くなる。
  - ソースコメントなどで補足。

Includeの実装



- 比較的簡単に実装可能。
- 扱いやすい単位にファイルをわけて管理できる様になる。
- タイムスタンプをチェックして再読み込み。

- コード例

```
Include(“Ability.nut”);
Include(“Eenmy.nut”);

function Initialize()
{
    local enemy = Eenmy();
}
```

コールスタックダンプ

- Squirrel には強力なコールスタックダンプが実装されていて、原因の特定を非常に容易にしてくれます。
- print 関数を任意に差し替えが可能。
  - デバッガのターミナルに出力
  - スクリーンに出力（デバッガの無い環境でも見える様に）
  - またセーブメモリーにも保存。（バグレポートに添付してもらう）

```
AN ERROR HAS OCCURED [the index 'hoge' does not exist]

CALLSTACK
*FUNCTION [attack()] enemy.nut line [4]
*FUNCTION [update()] enemy.nut line [11]
*FUNCTION [update()] enemy.nut line [14]

LOCALS
[a] 0
[this] TABLE
[i] 0
[this] TABLE
[this] TABLEc
```

ランタイムエラー

- Squirrelを使った開発で一番の問題だった点は、
  多くのエラーがランタイムに発生することです。

- これは動的型であるため、C++の様にコンパイル時に
  シンボルチェックを行えないため、問題があると
  ランタイムにエラーが発生します。

- また同様の理由で、コーディング中に
  作業割り込みが入ると書きかけのコードが
  バグ化しやすいといった問題もありました。

ランタイムエラーの原因

- タイプミスなどによるシンタックスエラー。
  - Windows 用にビルドした、sq.exe を使って
    事前にシンタックスチェック。
- シンボル（変数や、関数）が無い。
  - データシンボルに関しては、簡易的なシンボルマッチによる
    チェックツールを自作。
- 引数の型が一致しない。（C++バインドの関数）
  - 次のページで

引数型エラーの対策

- Squirrel上の関数の引数は動的型だが、
  C++バインド関数の引数には型があるので注意。

```
void test( float val )
{
            // 処理
}
```

```
 21|        local i = 10;
 22|        test( i );
```

```
AN ERROR HAS OCCURED [Incorrect function argument]

CALLSTACK
*FUNCTION [test()] d:¥temp¥floatInt.nut line [22]

LOCALS
[this] TABLE
```

- 型が合わないとランタイムエラーになります。

リソースのId参照

- Squirrel上でポインタを不正参照してしまうと
  原因の特定が困難になるため、原則的にはC++側の
  リソースの操作には id を利用。
- また、C++ 側では不正なid をチェックする様にした。

```
void WorldManager::SetActorPosX( int id, f32 x )
{
    SQUIRREL_ASSERTMSG( m_mapObject.find( id ) != m_mapObject.end(), "id=%d", id );
    m_mapObject[id].SetPosX( x );
}
```

メモリまわりについて

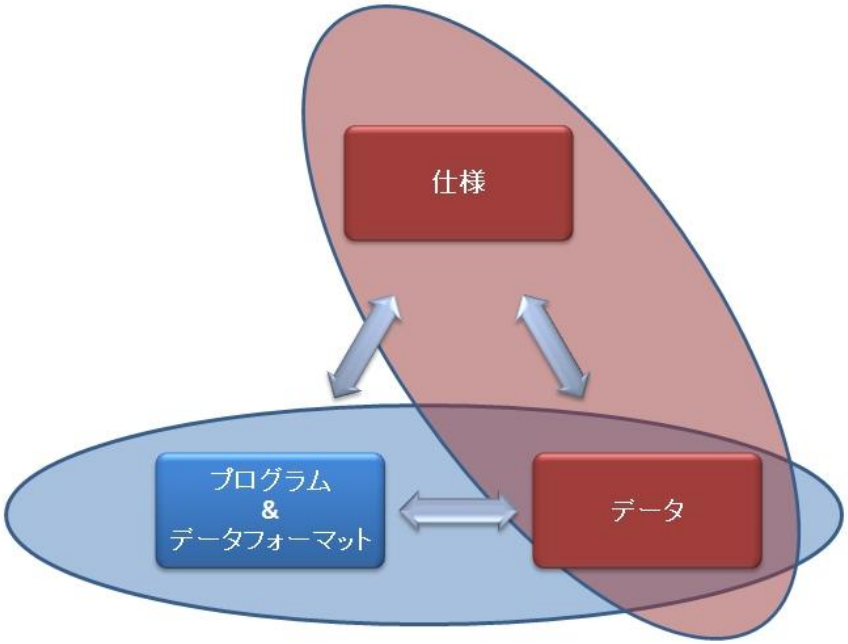
- Wiiの高速メモリ24MBのうち一部を除いて
  使用可能要領のほとんどをSquirrel専用に割り当てた。
- Squirrel だけで、50000件を超えるヒープブロック。
  - ヒープブロックオーバーヘッドサイズだけでも数ＭＢ。
  - ヒープのサーチ負荷の増大。
  - 簡単なテクニックと最適化により問題無いレベルまで軽減することができた。
- その他の表示リソースなどは独立してメモリ管理。
- Squirrel 上の String は realloc を多用するため
  メモリ負荷が問題になる。
  - String の操作はC++側に専用のI/Fを作成。

GC（ガベージコレクタ）

- 任意に呼び出しが必要。
  - なにもしないとメモリが無くなって止まります。
- SquirrelのGCはかなり優秀だけど、毎フレーム動かすにはそれでも重い。
  - ヒープブロック50000件も影響。
  - 実行タイミングに工夫が必要。
- さらに保険も必要。
  - GCが間に合わず、alloc や new が NULL を返す様な場合など。

プランナーのワークフロー

- プランナーが仕様を作成。
- プログラマーが仕様を元に、プログラムとデータフォーマット作成。
- プランナーがデータを追加変更してゲームを組み立てていく。



柔軟なデータフォーマット

- 敵データ例

```
::gameData.enemyDataArray <-
[
    {
        label=“worm”,
        vitality=4,
        strength=3,
    },
    {
        label=“moo”,
        vitality=3,
        strength=4,
    },
];
```

- 敵クラスの例

```
class Enemy
{
    _enemyData = null;

    constructor( id )
    {
        _enemyData = ::gameData.enemyDataArray[ id ];
    }

    function GetVitalityMax()
        return _enemyData.vitality;
    }
}
```

柔軟なデータフォーマット

- 敵データに abilityArray を追加

```
::gameData.enemyDataArray <-
[
    {
        label=“worm”,
        vitality=4,
        strength=3,
    },
    {
        label=“moo”,
        vitality=3,
        strength=4,
        abilityArray=[ ::ABILITY_BITE ], //追加
    },
];
```

- テーブルに abilityArrayが有るかどうか判定して処理を書くことが出来る

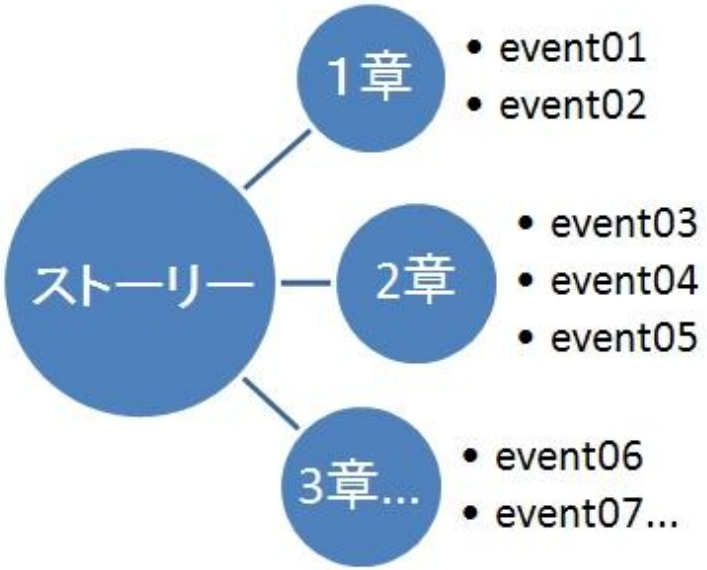
```
if( "abilityArray" in _enemyData ){
    // アビリティありデータ
}
else{
    // アビリティなしデータ
}
```

Excel -> XML -> .nut

- ２次元のデータについては、
  Excel を使って編集し、.XML を出力。
  自作のコンバータで .nut に変換する。

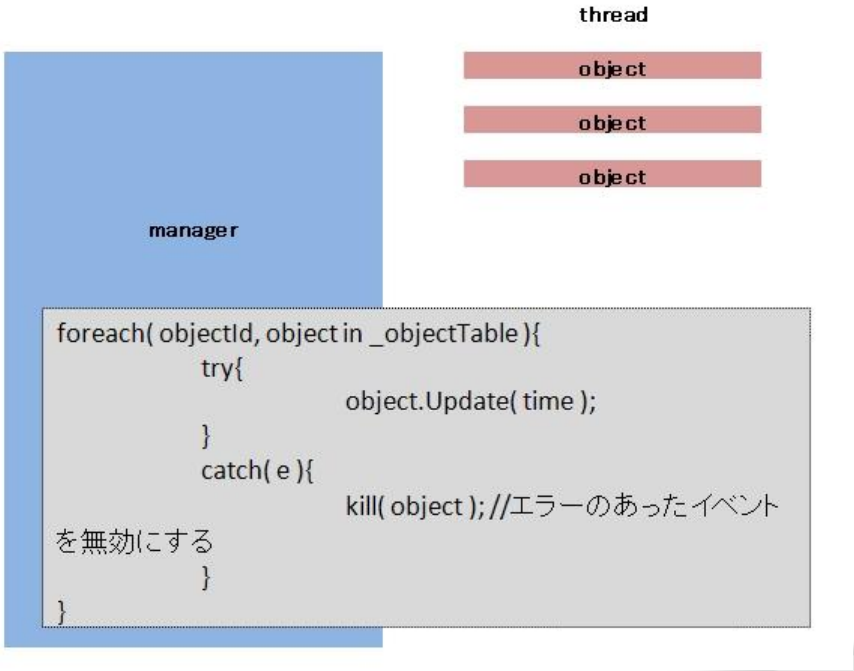
イベントシーンのワークフロー

- イベントのスクリプトも Squirrel で作成されていて、イベントスクリプト用にまとめた専用のI/Fを使って書かれています。



例外を使った復帰処理

- イベントは処理をオブジェクト単位で管理している。
- Squirrel の例外を使って、エラーが発生した処理オブジェクトを切り離し、エラーが発生したファイルをその場で修正し、再読み込みを行い処理を続行することが可能です。



Squirrelを使った開発のまとめ

課題

- ランタイムエラーとそれに伴う、デバッグコスト増。
- メモリの使用量やオーバーヘッドに
  気を使う必要がある。
- また完全なメモリマップの掌握が困難。
- C++側からSquirrelへの参照はスタック周りが
  若干ややこしい。

利点

- 実行中のコード編集と再読み込みを実現出来る。
- 柔軟なデータフォーマットが手に入る。
- 強力はコールスタックダンプによりバグの特定が容易に。
- C++コードよりも比較的簡単に書けるため、変更や、作り直しをより気楽に行える。
- メモリー破壊バグが、ほぼ0だった。
- メモリー周りに気をつければコードオーバーヘッドは、ほぼ問題にならなかった。

- 余談
  - elfサイズが大きくならないため、オーバーレイなどを駆使しなくてすむ。

結論

Squirrel かなりよいですよ！
今後も役に立ちそうです。

Q&A

```
::print("Thank you!¥n");
```