スクールガールストライカーズの内製クライアントエンジン

株式会社スクウェア・エニックス

杉本浩二 sugimoto@square-enix.com

21, June 2014

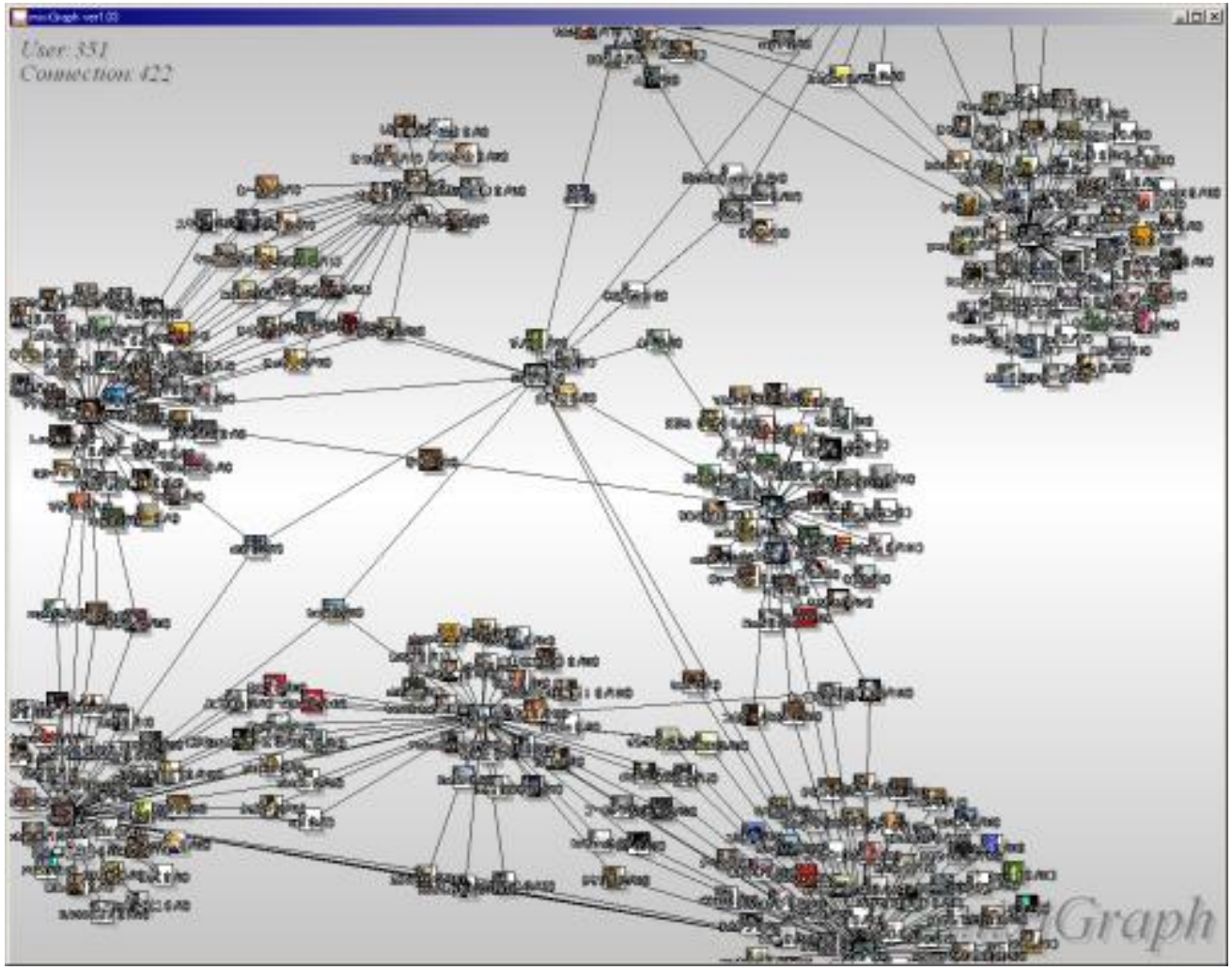
自己紹介

- 杉本浩二
- 1993年スクウェア入社 21年目
- 最初はクロノトリガー(SFC)から
- FF10のメインも担当
- CEDEC2012で内製ローカライズツール発表
- クライアントに限らずサーバもDBもハードも
- いろいろやるよ

デュープリズム

最初で最後のディレクション作品(1999年)

mixiGraph



覚えてる？

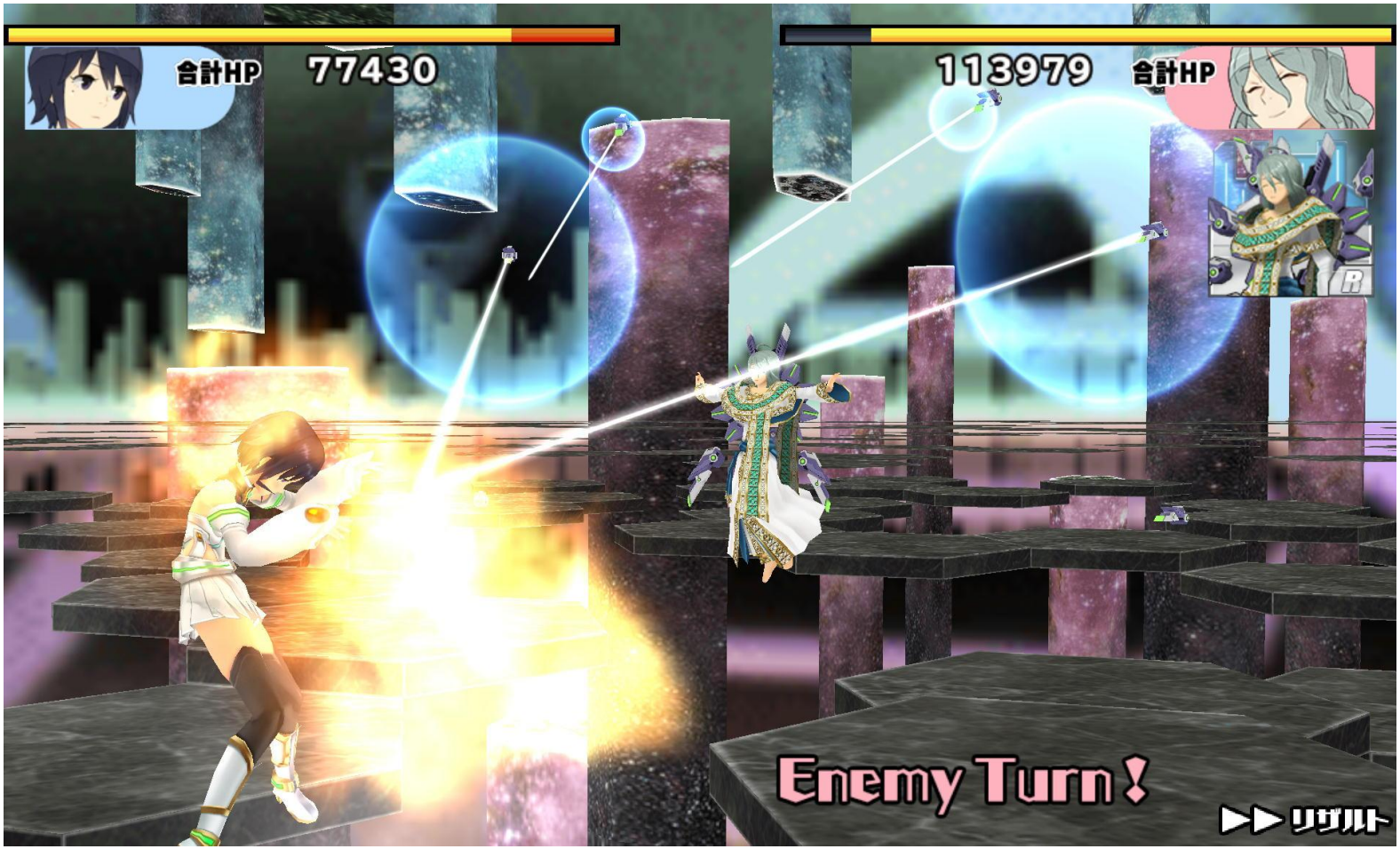
スクールガールストライカーズ(スクスト)について

- 3Dギャルゲー
- スクエニ社内開発
- iOS/Android™リリース済
- 今の所大きなトラブルも無く
- おかげ様で好評

HOME

ミッション

バトル1



バトル2

イベント1

イベント2

イベント3

今回お話する内容

- 1. 内製エンジンの紹介
- 2. コード生成
- 3. CPUキャッシュについて
- 4. リソースロードの仕組み

1.内製エンジンの紹介

特徴。

軽い

小さくて高速

iOS版は3.5MBytes

無料

カテゴリ: ゲーム

更新: 2014年4月17日

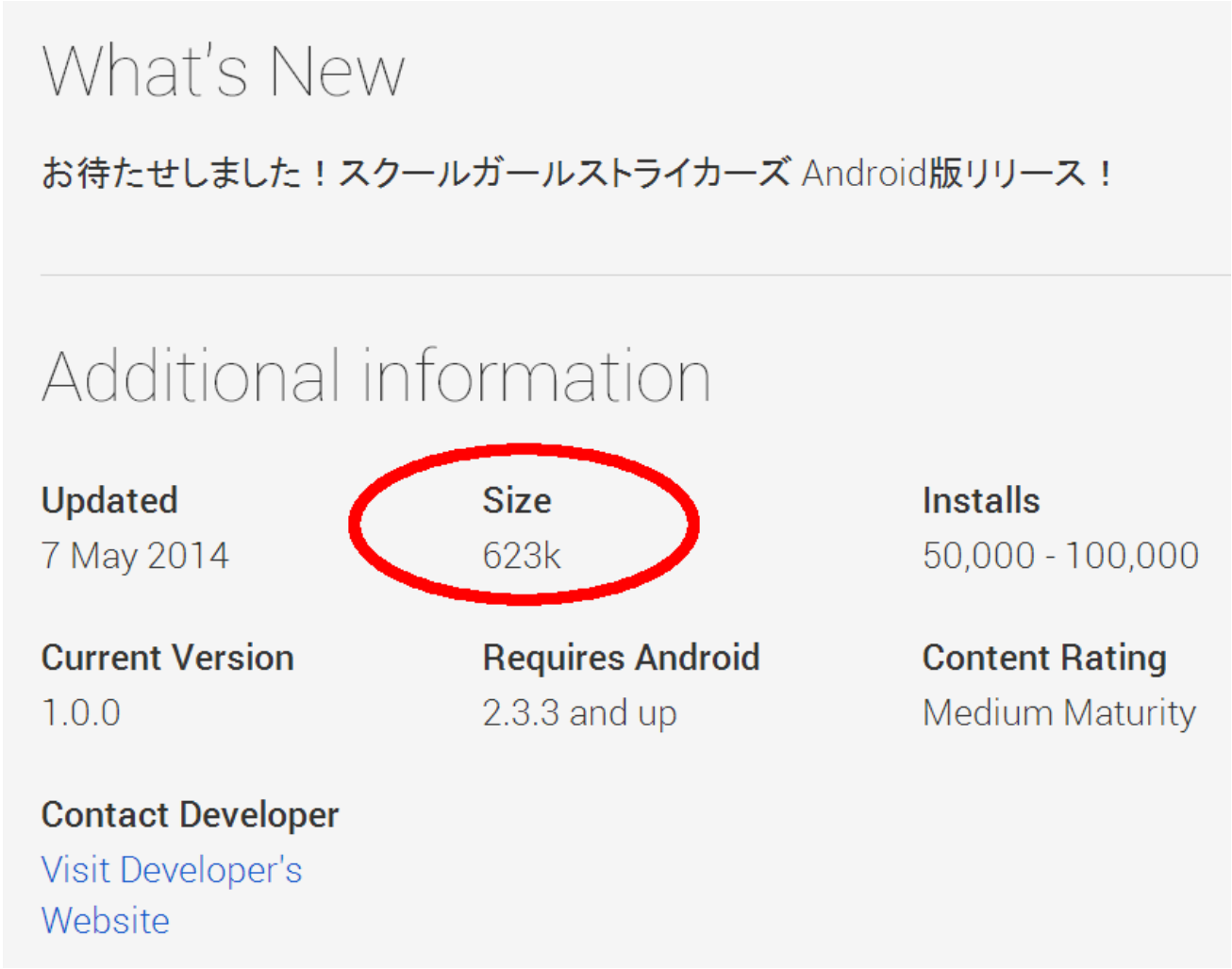
バージョン: 1.0.2

サイズ : 3.5 MB

言語: 日本語

販売元: SQUARE ENIX Co., Ltd.

Android™版は623KBytes



軽いと

- ダウンロードがすぐ終わる
- 起動が早い
  - 気軽に起動できる
  - →継続率が高まる
- 快適
- バッテリーが長持ちする

継続率が高まる

- ビジネスにするために、我々技術者ができる事
  - 「安定して正しく動く」は勿論
  - プロならその先の品質を
  - より快適に
- 低品質なゲームはプレイヤー離れに
  - ×バグる
  - ×落ちる
  - ×ロードが長い
  - 改善していきましょう

大事な事なのでもう一度

OPデモを実行しながらデータダウンロード

3D描画

杉本は3D得意

- PlayStation®最初期から3Dを担当
- ゼノギアス(PlayStation®)
- デュープリズム(PlayStation®)
- FF10/FF10-2(PS2®)
- 半熟英雄vs3D(PS2®)
- FF7 TECH DEMO for PS3®
- Crisis Core FF7(PSP®)
- The 3rd Birthday(PSP®)
- FF零式(PSP®)

そんな3D技術を現部署に持ち込み

- 「スマホで3Dやりましょう」
  - エンジン作りたかった
- 3Dに強いデザイナーもいた
- コスト試算→2Dゲーム並
- 差別化もできる
- →スクストの3D化決定

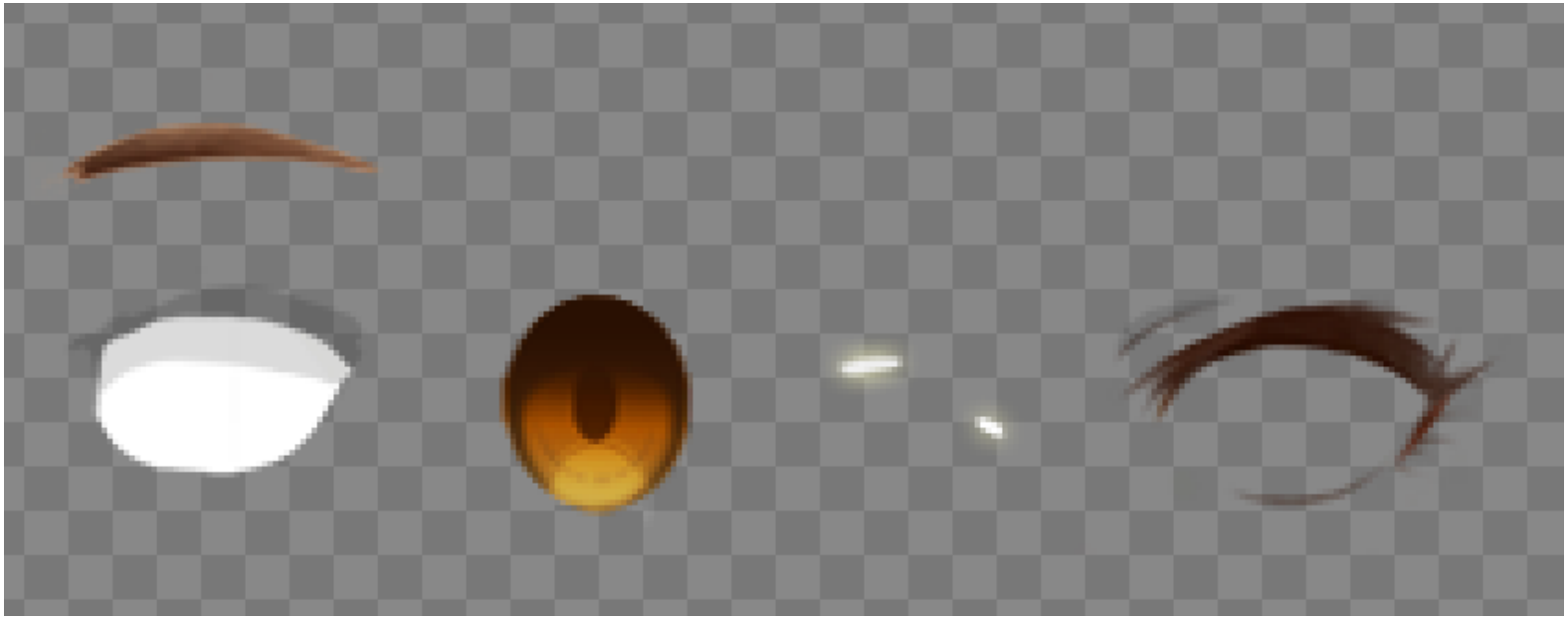
同時に5体の3Dキャラが

一体6000ポリゴン

60fpsでなめらかに演技する

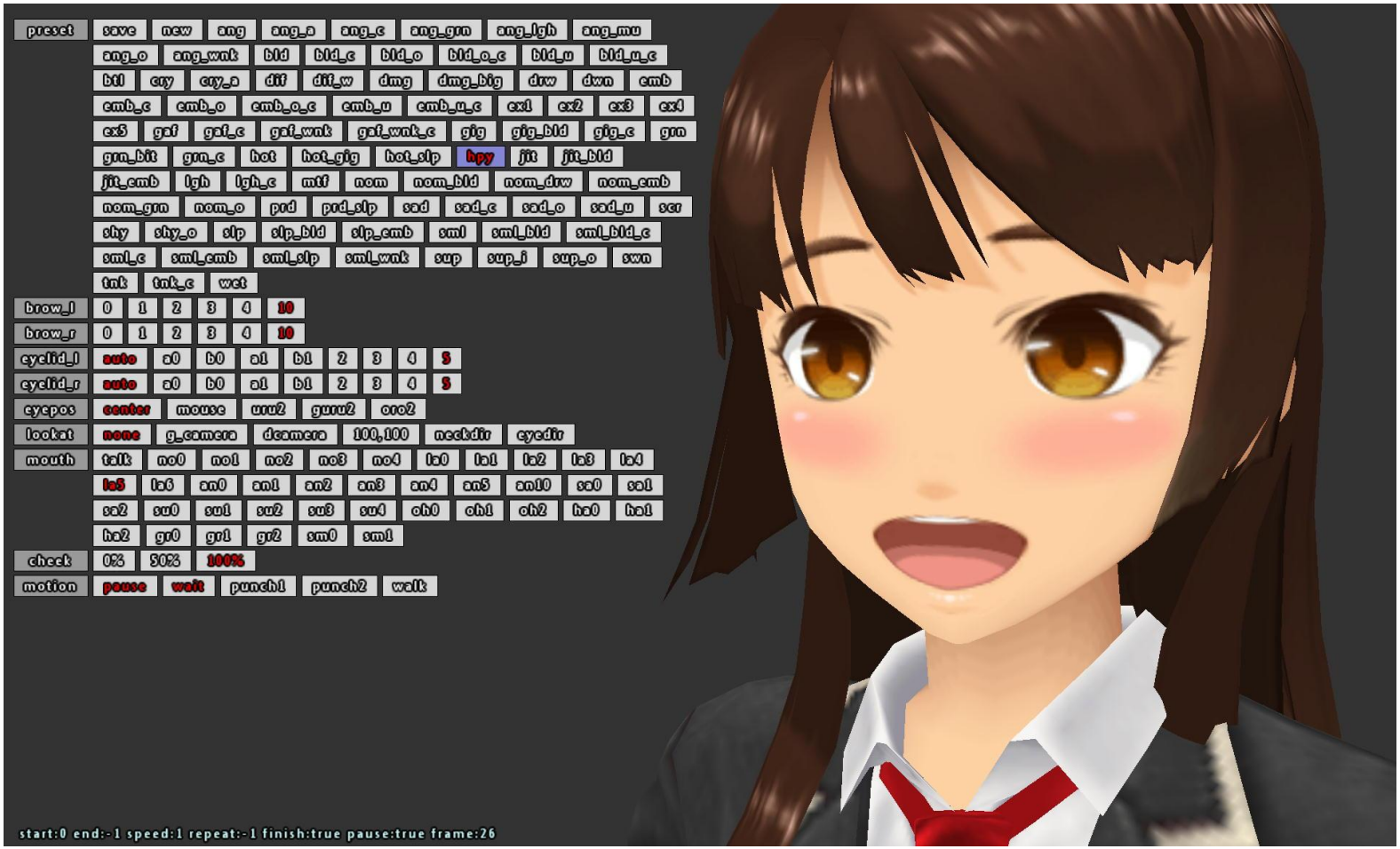
iPhone®4s基準

眉,白目,瞳,ハイライト,瞼をリアルタイム合成

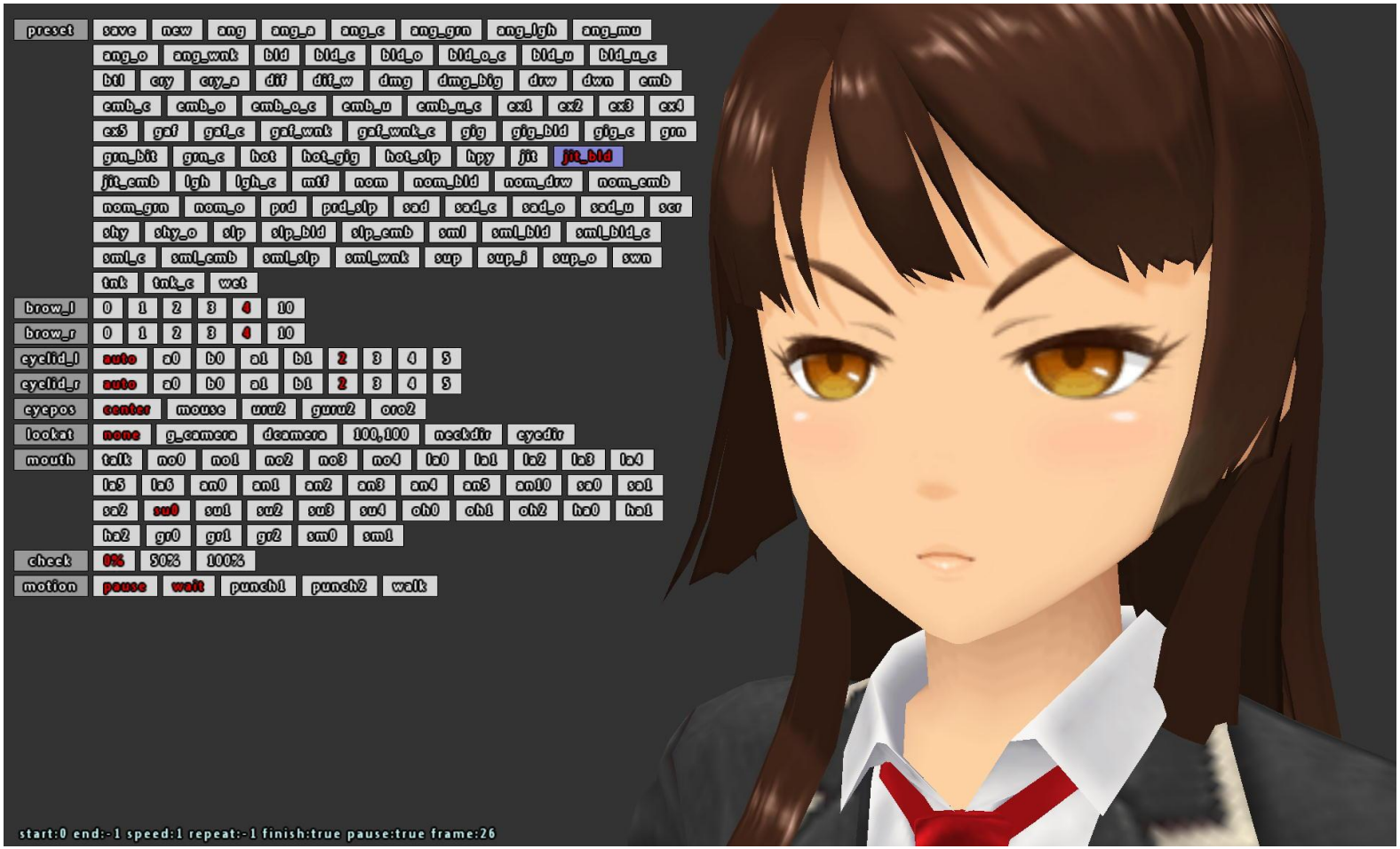


デュープリズムで実績のある技術

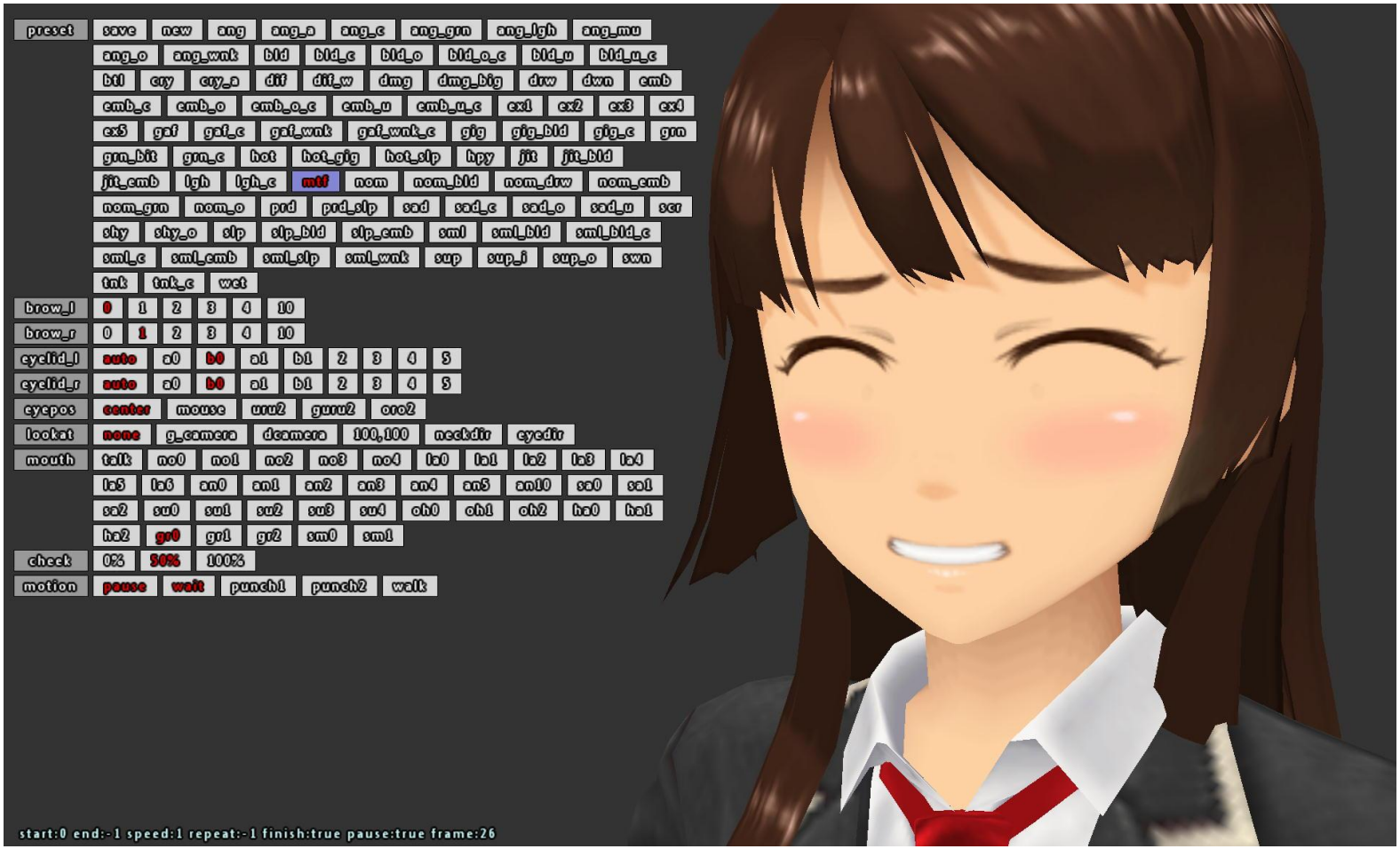
顔1



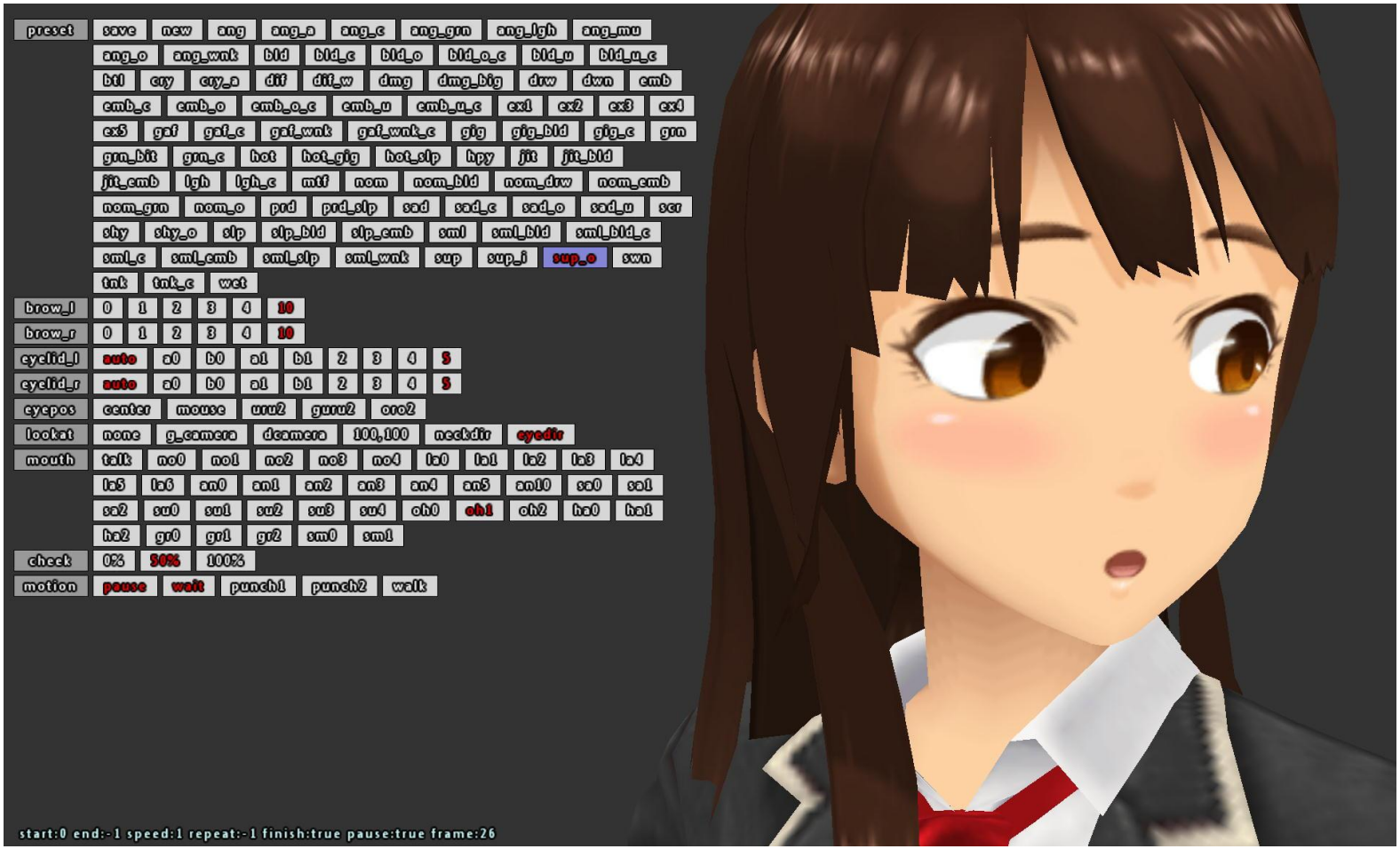
顔2



顔3



プログラマブルな視線制御



目線と首の角度制御

髪揺れ等

モーションではなくリアルタイム計算

仕様

- OpenGL ES 2.0
- C++で書かれている
- 多機種対応
  - iOS/Android™でリリース済
  - Windows®上で制作
  - 他プラットフォームへも移植中
- ビルド環境
  - Pythonでコンバータ、時々C++
  - JenkinsでCI
  - Mercurialでプログラム管理
  - Subversionでリソース管理

スクリプト言語「Squirrel」で制御

- http://www.squirrel-lang.org/
- Luaを使いやすくした言語
- 文法に癖がなく馴染みやすい
- 高機能
  - ゲーム開発には十分
- 小さい
- 速度はそこそこ
  - 速くは無い。遅くもない。
- 動的型
- スタックマシン

フルスクラッチ開発

- Cocos使ってないよ(ごめん)
- コンシューマ開発の経験を元に最初から設計した

2.コード生成

従来スタイル開発の限界

- 膨大なインターフェース関数群
  - コーディングはもちろん
  - ドキュメントも
  - wrapperも
- 一人で全部書くのは大変
- メンテも大変
- .cppと.hを別々に書くのはナンセンス

プログラマーならプログラミングをプログラムしようぜ

- 自分の代わりにプログラミングしてくれるプログラムを開発
- 設計を入力するとソースコードを出力する
- 楽ちん
- 冗長な作業は人間の仕事じゃない

今回のコーディングルール

- クラスメンバへのアクセスは関数で統一
- 関数オーバーロードでsetter/getterを区別
  - rue.rotation(mint.rotation())
- メソッドチェーン化
  - setterは自身のinstanceを返す
  - mint <- Mint().load().position(...).rotation(...)

yaml

- 構造化されたデータを記述するためのテキストフォーマット
- 人間が書きやすい
- (このスライドもyamlからpythonで生成)

設計をyamlで定義

- 型も名前も
- 引数の仕様も
- 例外条件も
- ドキュメントも
- 実際のコードも
- 自動化できない事は全て

pythonで処理

- 簡単な構文解析
  - 文字列処理レベル
- 依存関係の調査(暫定)
- 情報をpython object化
- 必要なコードを生成して
- ソースを出力

たった2行の定義から(.yaml)

```
class Object:
    Position position:
```

C++ソースコード生成(.h)

```
class Object{
	Position _position;
public:
	inline const Position& position() const;
	inline Object& position( const Position &p );
	inline Object& position( float x, float y, float z );
	inline Object& position( const Animation<Position> &a );
};
#include "object.inl"
```


型によってsetterのオーバーロード関数を増やしていく

C++ソースコード生成(.inl)

```
inline const Position& Object::position() const
{
	return _position;
}
inline Object& Object::position( const Position &p )
{
	_potision=p;
	return *this;
}
inline Object& Object::position( float x, float y, float z )
{
	return position( Position( x, y, z ) );
}
```

Squirrelへのバインド関数生成(.cpp)

同じ名前の関数をまとめて、Squirrelでのオーバーロードを実現

```
static SQInteger Object_position( HSQUIRRELVM v )
{
    Object *v1;if( sq_getinstanceup( v, 1, (SQUserPointer *)&v1, Tag_Object )==SQ_ERROR )return SQ_ERROR;
    SQInteger nargs = sq_gettop( v );
    if( nargs==1 ){
        *reinterpret_cast<Position *>(create_instance( v, Tag_Position ))=v1->position();
        return 1;
    }else if( nargs==2 ){
        SQObjectType type = sq_gettype( v, 2 );
        if( type == OT_INSTANCE ){
            Position *v2;if( sq_getinstanceup( v, 2, (SQUserPointer *)&v2, Tag_Position )==SQ_ERROR )return
 SQ_ERROR;
            v1->position( *v2 );
            sq_push( v, 1 );
            return 1;
        }else if( type == OT_TABLE ){
            return entry_animation( v, (void *)v1, OBJECT_POSITION );
        }else{
            return sq_throwerror(v, ERR_INVALID_TYPE );
        }
    }else if( nargs==4 ){
        SQFloat v2; sq_getfloat( v, 2, &v2 );
        SQFloat v3; sq_getfloat( v, 3, &v3 );
        SQFloat v4; sq_getfloat( v, 4, &v4 );
        v1->position( v2, v3, v4 );
        sq_push( v, 1 );
        return 1;
    }else{
        return sq_throwerror( v, ERR_WRONG_NUMBER_OF_PARAMETERS );
    }
}
```


人間は1文字も書いて無い。

他にも自動生成

- 普通に関数定義も
- ドキュメント
- 引数チェック
- 遅延評価コード
  - 依存関係から、不必要な計算の回避
- シリアライズ関数
  - メンバ変数を集めてjsonにしたり
- コンストラクタ用初期化
- 計測コード挿入

プログラムから設計にアクセスして、なんでも生成できるようになった

修正が楽

- 生成パターンを一カ所修正すれば、同じパターンのコードがまとめて更新される
  - 数百ヶ所の修正も一瞬
- リファクタリングしやすい
- チューニングしやすい

同じコードは二度書かない

- プログラミングを半自動化
- ルールから外れた実装は存在できない
- 処理速度と生産性と安全性を確保できた
- 一般的なプログラミングスタイルではないのでリスクになるかも

1人だからできた

3. CPUキャッシュについて

よくある勘違い

- ×キャッシュメモリは速い
- ○メインメモリが遅い

計算するのはCPU

メインメモリは超遅い

- L1キャッシュの数百倍の時間がかかる
- キャッシュミスは日常的に発生している
  - この頻度を下げたい
- アクセスには電力が必要
  - バッテリー消費の原因に
- CPUに本来の性能を発揮させるために
- キャッシュから外れない努力を

CPUメーカーも凄く頑張っているので、昔と比べたらミスしにくいけど

iPhone® 4sのCPU A5

- ARM Coretex-A9
- L1 cache
  - instruction 32KB / core
  - data 32KB / core
- L2 cache
  - 1 MB shared
- 2-way set-associative
- 64bytes line

®

大きなプログラムは悪

- プログラムが大きくなるほどキャッシュから外れやすくなる
- 数百倍遅いメインメモリへのアクセスが頻発
- 負の連鎖

このエンジン

- 常駐コードサイズは約600KBytes
- 全体の5%がL1のi-cacheに足をかけつつ
- L2 cache(1MBytes)に全部入る大きさ
  - データや他プロセスと奪い合っているので実際に全部入っているわけではない
- 動作が軽くてバッテリー消費が少ない大きな理由

キャッシュに入れましょう

コードサイズを小さく

- メモリが豊富な時代になって、サイズを気にしないプログラマーが本当に増えた
  - そうじゃねーだろ
- 重複コードの徹底排除を
- 大規模なライブラリの利用には注意
- 美しいコードは自然にシンプルになる
- 基本的な事

ループを小さく

- ループはキャッシュサイズを超えないように
  - まー大丈夫でしょう
- ARMの動的分岐予測は大変優秀
- 64Bytes未満のループでは、CPUが覚醒する
  - Fast Loop Mode
  - i-cacheへのアクセスすらしない
  - 消費電力もさらに抑えられる
- 小さくてキツいループは分割してみては？

積極的なinline展開

- サイズを犠牲にして速度を稼ぎたい時にinline展開を使う事があるが
- 最近のコンパイラはとても賢い
  - 初めてアセンブラを書かずに終えた
- callableな関数はオーバーヘッドが大きく、inline化した方が小さくなる事が結構ある
- 一つの関数はできるだけシンプルに
- あとはコンパイラ先生が良い仕事をしてくれる

純度の高いコード生成を

- CPU様に対して人間都合の不純物を残さない

4.リソースロードの仕組み

スクストはロードが早いと
評判らしい

ディスクデバイス時代の苦悩

- 容量は多いが、基本的に遅い
  - リードが遅いだけならなんとかなるが
  - シークが遅いのは厳しい
- 設計の大きな足枷になっていた
  - 読み込み順序を調整したり
  - まとめたり
  - 重複させたり

スマートフォンはメモリデバイス

- 回転する円盤からの解放!
- 高速になりました！
- 設計が楽になりました！
- シーク時間が無くなったのが有り難い

とはいえ

- 読み込み時間は０ではない
  - 体感できる程度には遅い
- 現状に甘えず品質を上げる
- PC用のお下がりではなく、スマホ専用の設計で
  - 小は大を兼ねる
- 不必要な処理を排除して高速化

OSのファイルシステム

- 1アクセス最低でも512Bytes転送される
  - どんなに小さなデータでも
- 開く時
  - OSがファイルを検索
    - →ディレクトリ構造を辿って数回アクセス
- 読む時
  - 読み込みサイズが未定
    - →一度SEEK_ENDしたり
    - →何度かに分けて読んだり
- 1つのファイル取得には、多数のアクセスが発生
  - 多少はメモリにキャッシュされているけども

アクセス回数を減らしましょう

スクストの仕組み

はじめはsqliteを使ってみた

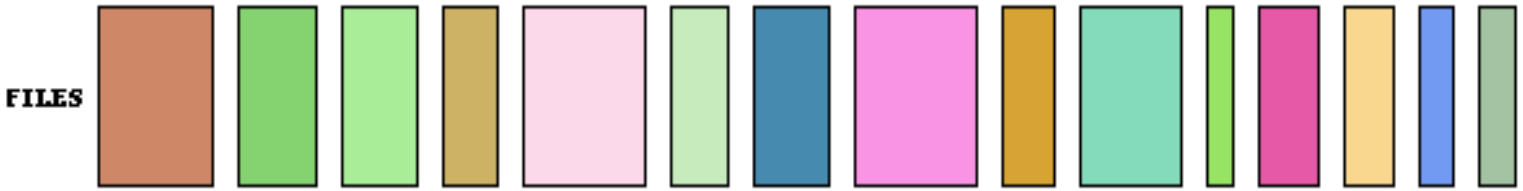
- 更新は楽
  - ただし遅い
- 当時700KBytesのエンジンに対して500KBytesも増えた
  - いろいろ削ったけど、そこまでして使いたくない
- indexingはB-Tree
  - アクセスが頻繁に発生する
- 要求に対してオーバースペックすぎ

結局自前で作った

- サーバからのパッチによってリソースが追加・削除・更新される
- 必要なデータをリクエストして返って来るだけでいい
- 別スレッドで動作

元リソース

- スクストは現在約1万2千リソース
- 全てのリソースに32bitのidを設定
  - これがファイル名代わり

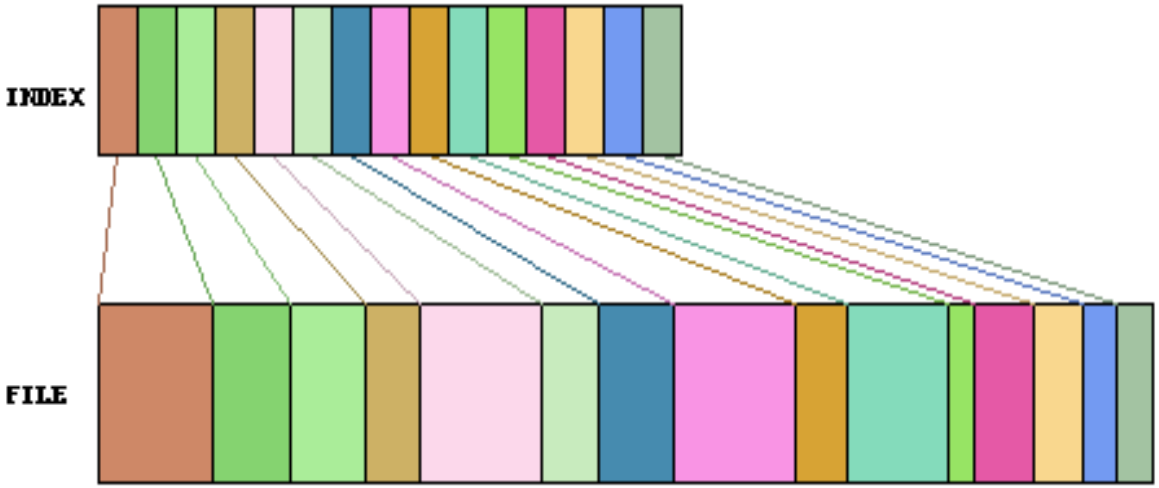


すべてのリソースを結合

- 約240MBytesのファイル1本に
- ゲーム中はずっとopenしている

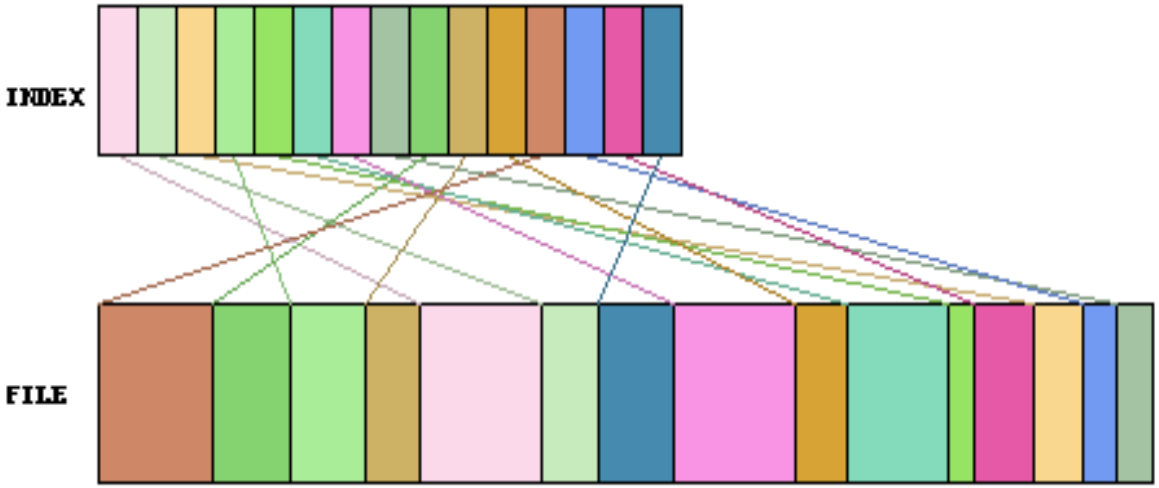
インデックス配列はオンメモリ

- 小細工無し、ただの配列が1本(id,offset,size,...)



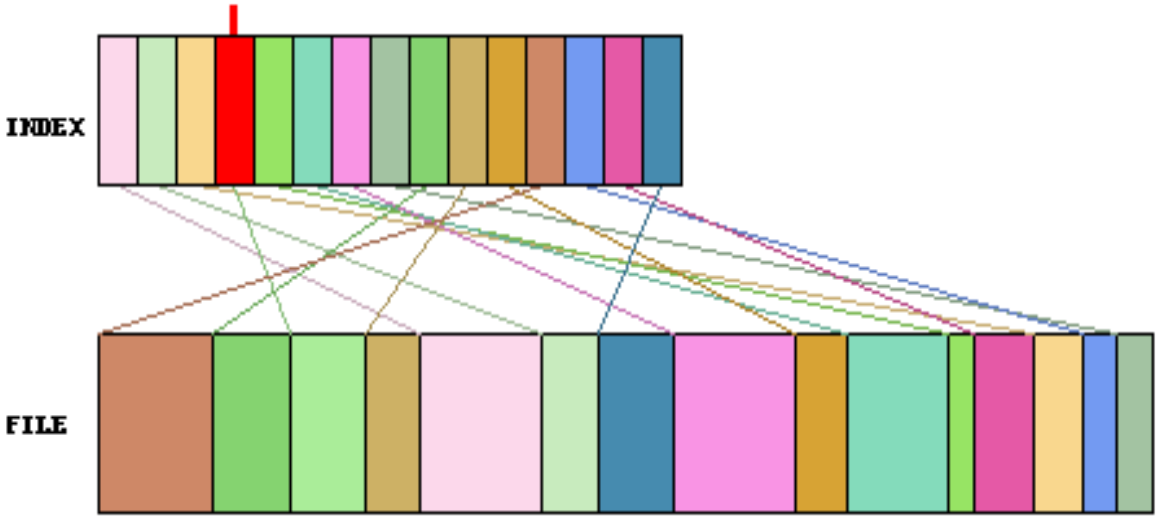
インデックス配列はオンメモリ

- 小細工無し、ただの配列が1本(id,offset,size,...)
- id順でソート済



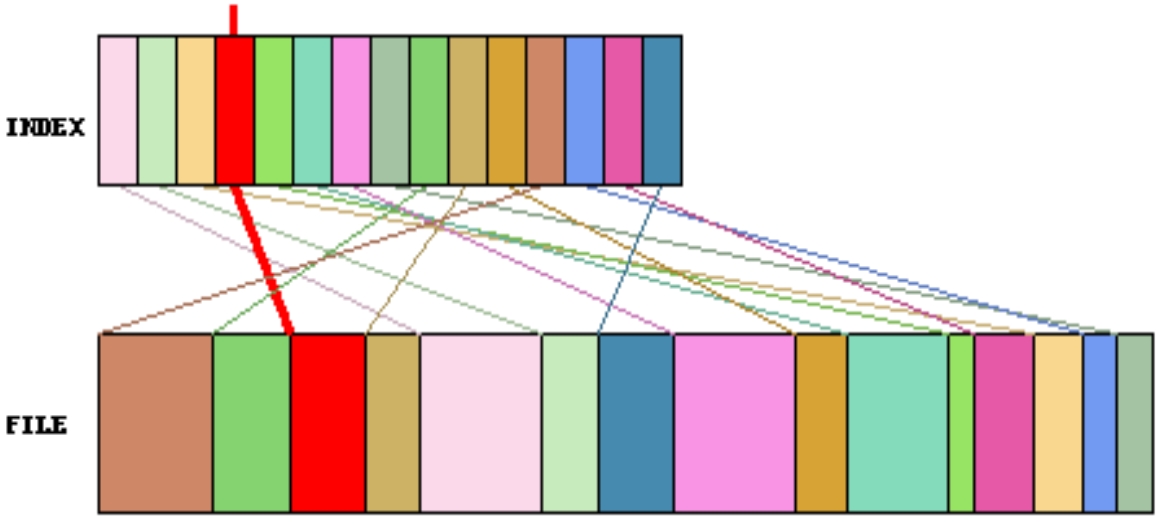
インデックス配列はオンメモリ

- 小細工無し、ただの配列が1本(id,offset,size,...)
- id順でソート済
- idを与えるとバイナリサーチ
  - offsetとsizeが返る



一発seek、一発read。

- 1リソース1回のアクセス
- 超シンプル
- かつ高速



これだけ

- シンプルが最強

ディスク時代の技術も

- 先読み
  - 当然
- メモリキャッシュ
  - 昔ほどではないが、効果はある

最後に。

これから

- このエンジンを拡張して
- 他のタイトルでも
- 利用していく予定

スクストのダウンロード

- 公式 http://schoolgirlstrikers.jp/
- iOS版 App Store℠からダウンロード
- Android™版 Google Play™からダウンロード
- インストールは一瞬
- このスライドで紹介した技術を
- 実際に使っておりますよ

おわり

sugimoto@square-enix.com

@k_sugimoto