

TSUCHINOKO PRESS

なんかカッコいいゲームを支える技術

カッコいい演出とかをするやつ

川野 竜一

[著]

「とにかく完成させる」で終わらせない。速く、カッコよく、ド派手に。

DxLib・DirectX・Vulkan

シェーダ・インスタンス・パーティクル

2Dをカッコよく・3Dをカッコよく

頂点の扱い方・オクルージョンカリング

ド派手なパーティクル・画面揺れ

つちのこ技術社

はじめに

はい、ちょっと適当に表紙を作りましたが本当に適当なので、この通りに授業が進むわけではなく(そんな時

間、あるはずない)、第一目的は皆さんが就活するまでにゲームがきちんと完成させていることが第一目的です。

この授業ではそれをちょっと派手にしたりするための Tips 的な感じでやっていくつもりです。

Tips ということでどうしてもネタがとりとめなくなってしまうため、皆さん各自で『これを解説してほしい』などのネタがございましたら提供をお願いします。

で、ネタですが、何でもいいです。プラットフォームも言語も問いません。

あとこの授業は次年度就職年次なので正直時間がございません。スケジュール等に関しては最初に説明があるかと思います。

まあ大変かもしれませんが、次年度就職のこの時期は仕方ないと割り切ってください。既に今までの ASO GAMESHOW とかできちんと完成した作品を作っている人はスケジュール感とかが分かってるとか、どこに修羅場が来るのかなんてのが分かってるかなと思いますが、それはごくごく一部で(クラスの2割くらい?)、毎年の傾向として大半の学生さんは…

最初は張り切って詰め込みまくって、中盤に中だるみしてしまったり、後半に焦って何とか完成させる(もしくは完成しない)ってのが大半かなと。

フランス人から見た日本のゲーム開発

↑に見られるようにどうやら日本人にありがちなことらしいので、常に意識してないとその罠にはとられてしまうっぽいので、1 週間ごとに自分でもチェックし、他人からもチェックしてもらうというのが必要かなと思います。

この場合の『何とか完成』ってのがだいたい『α版』って感じで、クオリティが低い…完成度が低すぎるものが多く、自分でもそれ(完成度低いの)が分かっているのに結局就活でいつまで経っても作品を送って就活しないということになります。

なのでスケジュールいっぱいいっぱい使って完成させるのではなくて、『12 月頭(もしくは 11 月末)でβ版』を目途になるようにします。そうすると 11 月上旬くらいにα版ということになるため意外ともうギッチギチのスケジュールになります。その覚悟は今のうちにしておいてください。

遊ぶ暇がない…というのは受験生と同じだし、別の学校の『ガチ国家試験(公務員だの整体師だの)』の試験前は同じなので、きつかるうがどうだろうが人生のうちの多少きつい期間のうちの1つだと割り切って頑張ってください(学校がマシなのは、お助けの仕組み[友人とか先生とか]がある程度あるということです…が、夏休みの宿題が残り 1 日の状態で泣きつかれてもお父さんもお母さんもどうしようもないのと一緒に、末期になる前に助けを求めてください)

さて、完成版が早めに出来たらあとは遊んでればいいかということそういう事じゃなくて『完成度』を上げていきます。

最近のゲームの流れが近いのですが、リリースしてから更新更新をかけていって、より良いものにしていくイメージですね(昔のゲームは売り切りなのでこれができなかったですが…)

まあぶっちゃけこれが『苦痛』とか『できない』とかって人はゲーム会社に向いてないかなと思います。いくら労働環境が良くなったからと言って、作ることに情熱がない人は…なんかの間違いでゲーム会社の入っても続きませんので……。

ですから、『もしゲーム会社入る気はもうないから『解放』してくれ』って人は担任等にご相談ください。



俺はあまり企画の話はできないけど…

ゲームを作る要素としては『ルール』『状況/環境』『ビジュアル/サウンド』なんだろうけど、『自分が面白い』と思うゲームは何か？それをなぜ面白いと思ったのかを基本的な要素にまで分解しまくってください。その過程で何に対して『面白い』と思ったのかをピックアップして、それをゲームのルールや環境に適用してください。

それはグラフィック的な部分に関しても同じですね。何をカッコいいと思ったのか。

ただ、どれでもそうですが、かなり見落としがあると思いますので、要素を出し切った後で、そこで終わりじゃなくて、『本当にこれで全部か？』と疑って見直してみましょう。

あとはそれに点数をつけていって、メインの要素はなんなのか、そういうことを解析していきましょう。

そこで『おもしろいと思ったゲーム』の選定方法ですが、『遊び込んだゲーム』はやめといたほうがいいです。何故かということ慣れ過ぎて分からなくなってる(独りよがりになってる)事が多いです。なので、おすすめは『試しにやってみたら面白かった』ようなゲームを題材にするほうがいいかなと思います。

『はじめに』はここまでですね。じゃあ Tips とかやってみましょうか。

ちなみに Tips は週の最初のほうで、週末 2 日はみなさんのチェックをしていこうかと思います。あとはリクエストが

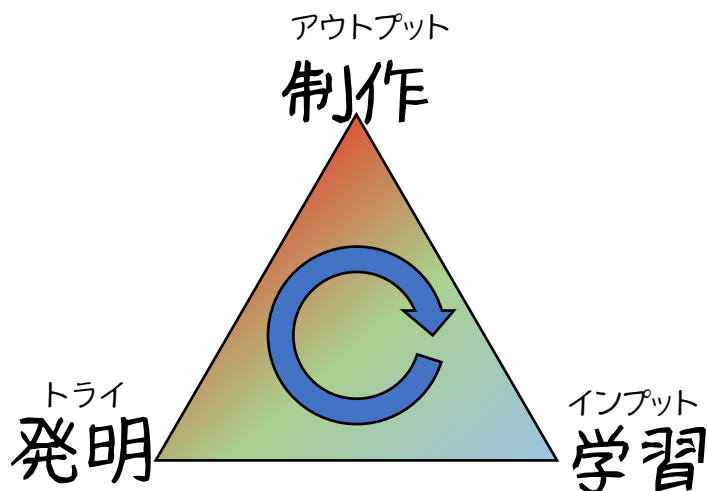
目次

はじめに	1
俺はあまり企画の話はできないけど...	3
お勉強 Tips	6
図書館を使おう	6
Web サイトを使おう	7
おすすめの書籍	8
アウトプット	9
発明(アイデア)	10
スケジュール Tips	11
バッファつきスケジューリング	11
実装レベルまで細かくスケジューリング	11
仕様書を作るときのコツ	13
仕様書の最初のコツ	13
仕様書にしていく工程	13
忘れやすい登場人物(オブジェクト)の例	13
仕様書とオブジェクト指向設計	13
ユーザーの存在を意識してね	13
プロトタイプ の作り方	14
ガントチャートの意義と構築について	14
プログラミング Tips	15
シェータについて	16
シェータ書く前にさ	16
レンダリングパイプラインについて軽く... ..	17
3D の知識の基礎について	19
スキャンライン法と射影変換	20
2D のシェータから挑戦しよう	23
DxLib におけるお勧めの『初歩シェータ習得までの流れ』	23
2D で 3D っぽくする(法線マップによるシェーディング)	25
お手軽に画面の賑やかしに使える『レイマーチング』	31
レイマーチング基本というか初歩の話	32
2D における距離関数	33
fmod で大量生産	33
3D における距離関数とレイマーチング	36

法線ベクトルを返してみる	38
球体以外の時の法線	39
その他背景にぎやかし活用法	43
輪郭線	44
スプライトアウトライン	44
オーラ	45
3Dアウトライン	46
動的な法線マップの作り方	49
設計とかについて	49
ダイアグラム(クラス図)を書いてみよう	51
サンプルクラス図	52

お勉強 Tips

最初なんて細かい話をしても仕方がないのでゲームを作るための勉強に関する Tips を話していこうかと思います。



なんでこういう図を書いたかというと、人によっては『これのどれかに囚われて動かなくなる』事が結構あるので、上のこの3つをローテするということを意識しておかないと結果につながりません。

次年度就職においては『制作』が最もウエイトでかいんですが、それでも他をおろそかにすると広がりもなくクオリティも低くなります。本当にしんどいですが、この3つぐるぐる回していくのを意識しましょう。

まずはインプットの話から

図書館を使おう

インプットと言えば書籍や Web サイトを読むことですね。

僕は図書館のヘビーユーザーなのでこれは毎年言ってるんですが、図書館を利用しましょう。で、最近の図書館は電子書籍貸し出しも行っていて、福岡県立図書館だと…

<https://www2.lib.pref.fukuoka.jp/>



こういうのがあります。『どーせしよもないのしか置いてないんだろ?』と思ったら

- リファクタリング
- ゲーム制作者になるための 3D グラフィックス技術改訂 3 版

などがあります。ちょっと細かくは自分で探してほしいのですが、意外とあります。ま、夕夕で読めるので

Amazon ほどのラインナップではないのですが、タタで読めるのは強いと思います。

あ、でも図書館利用カード作っとかないと使えないので、お近くの図書館に行って作っておきましょう。

ちなみに僕は福岡県立図書館と福岡市立図書館両方持っています。

<https://toshokan.city.fukuoka.lg.jp/>



こちらも電子図書はあるのですが、県立に比べると数学関連書籍やコンピュータ関連書籍が弱い気がします(基本は利用してません…物理書籍は強い図書館なのに…)

Web サイトを使おう

有名どころでは Qiita

<https://qiita.com/>

なんか一時期 Qiita じたいが炎上してにわかに勢力を伸ばしつつあるのが Zenn

<https://zenn.dev/>

言わずと知れた Cedit

<https://cedil.cesa.or.jp/>

SlideShare(Capcom R&D などが有名)

<https://www.slideshare.net/>

Speaker Deck(cygames の技術部とかが多い)

<https://speakerdeck.com/>

体系的に学ぶならちょっと古いけど「ゲーむつくるー」

<http://marupeke296.com/GameMain.html>

Project ASURA(レンタルリング寄り)

<http://www.project-asura.com/>

あと、他人のコードを読んでお勉強(まんまパクるなよ)

<https://github.com/>

あと、Unity やら UE4 なら Youtube に参考動画が多いかな

<https://www.youtube.com/>

あとは用途に合わせて Google 検索してほしいんですが、怪しい、嘘教えてるサイトもあるので、最終的にはベンダーのドキュメントを読んでください

Unity→Unity Technologies

UE4→Epic Games

DxLib→公式サイト

DirectX→Microsoft(たまにここが間違っていたりするんで気を付けよう)

グラボ→nVidia or AMD or ... Intel

おすすめの書籍

体系的にいろいろ学ぶなら書籍に勝るものはないかなと思います。で、ここでお勧めする書籍はプラットフォームにあまり依存しない、コーディングのための書籍です。

- **ゲームプログラマのためのコーディング技術**

- Game Programming Patterns

この2つはおさえておいたほうがいい(だけど図書館の電子書籍にはなっていないので、古本屋とかで探してみよう)

- Effective C++

- More Effective C++

- Effective Modern C++

基本の本ではありますが、ちょっとお堅いのが難点。前者2つやってればある程度はこの本の内容もカバーできるから最初の本2つがあればいいかな。

- Clean Architecture

基本の事だけど、意外と知らない事が載っています。プログラミングを行う際の『○○原則』などは必見でしょう。

あと、ここまでは綺麗なコーディングのためのものでしたが、実際の実装に関しては

- ゲームを動かす技術と発想

- ゲームを動かす数学・物理

あたりがわかってれば、初歩というか基本的なところは大丈夫だと思います。

数学に関しても、まあベクトル/行列までをしっかりと使いこなせてれば(使いこなせてればだけど)、十分です

(少なくとも学生の間は)。

そこから先に行こうとすると、確率統計や線形代数や微分積分などの考えが必要になりますが…今はよほど背伸びした研究とか実装とかしない限りは大丈夫でしょう。

あとグラフィック系で背伸びするなら

- ゲーム制作者になるための3Dグラフィックス技術

ですね。これは図書館電書になってるので、福岡県民は見れます。それぞれの自治体の図書館とか見ておいってください。

数学をもう少し背伸びするなら

- 実例で学ぶゲーム3D数学
- ゲームプログラミングのための3Dグラフィックス数学

とかいろいろありますが、もう残り時間も少ないので、スキマ時間等に最初にあげた2つに目を通しておくのがいいかと思います。

アウトプット

と、まあここまではインプットのお勉強。お勉強というのはインプットだけでは身につかない。とにかくアウトプットが大事。インプット3に対してアウトプット7以上くらい。

とはいえゲームを作ることアウトプットとするとあまりにもハードルが高いと感じるかもしれません。そういう時は、小さいテストプログラムで勉強内容や、重いつきのアイデアを形にしてみましょう。

<https://github.com/boxerprogrammer/TestCode>

シヨボくてもいいし、Privateでもいい。とにかくコードの形にしておくのが大切。

また、自分の書いたテストコードに関する説明とかも書いておいたほうがいい。他人に説明する準備や訓練にもなるし、何よりも自分の頭が整理されます。

できればQiitaやZennとかで記事を書いておけば、有用なものなら自分の名前を広める事にも役立ちますし。書き溜めれば『技術同人誌』として売ることも考えられます。

とにかく自分の頭の中をコードでも文章の形でもいいので外部化するのは本当に重要なので必ず実践してください。

発明(アイデア)

まあ砕けて言うと『思いつき』ですね。でもこの『思いつき』が結構重要で、画期的なアイデアや問題解決に結びついたりします。

とはいえ、これはコーディングしてたり作業してたりする時ばかりに思いつくわけではなく、とんでもないタイミングで思いつくことがあります。

なので常にちっちゃいメモ帳(手のひらサイズの防水のやつ)とか、小さいスマホ的な奴(僕は中古の ipod touch 使ってます)

思いついたものは時間が経つとゆっくり消えていきます。時間が経てばたつほど『あれ?なんだっけ?』になってもったいないので、走り書きでも何でもいいからメモっておくことです。

ちなみに『防水のメモ』なのは理由があって、風呂に入ってる時とか自転車に乗ってる時にアイデアが出ることが多いんですが、そういう時に濡れてもメモを書けるようにです。ipod touch などは ZipLock あたりで防水しとけばいいかなと。

あと、独り暮らしの人なら、家のトイレにメモ帳をぶら下げとくってのものありですね。トイレ時も思いつきが多いものです。

あと、退屈な授業や会議の時に授業を聞いてるふりをしつつノートの端っこにアイデアを書いておくのもいいです。眠たい授業や会議は時間の無駄なのですが、何故かそんな時は脳内で自分のアイデアが出やすいものです。

だいたい授業や会議の時はノートを持ち込んでいるので怪しまれずにメモ書きすることができます。こういうのを習慣化すると本当に財産になったりします。

自分が芸人になったつもりで、思いつきのアイデアは大事にしましょう。

で、このアイデアも形にしないとすぐ忘れたり、身につかなかったりするので、できれば外部化しましょう。もしかしたら思わぬ反響があるかもです。

スケジュール Tips

はい、ここではスケジュールのための Tips を書いておきます。前に『次年度就職年次だから多少無理してギリギリでやらなきゃいけない』とは書きましたが、不慮の事故やいろんなトラブルはつきものです。また、どうしても中だるみというのは発生します。

そういう事に対してどう対応するのかの Tips を紹介します。

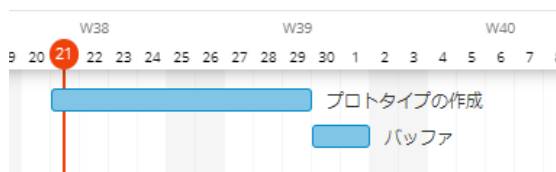
バッファつきスケジュールリング

皆さんは計画を立てるときに『ギッチギチのスケジュール』『すべてがうまくいく理想のスケジュール』にしているのでしょうか？

やめなされ、やめなされ。物事がすべてうまくいくとは限りません。風邪をひくかもしれませんしパソコンが壊れるかもしれません。ネットが使えなくなるかもしれません。

そうでなくても本当に色々と邪魔は入るものです。

そのためまず『理想のスケジュール』を立てた後は『バッファ期間』を設けておきます。



こんな感じですね。このように 1 つのタスクの後にバッファでもいいですしいくつかのタスクをまとめた後にバッファを置いてもいいです。

で、そのバッファを置いた状態で全体のスケジュールを見てみます。たぶんすべて詰め込もうとしたらもとのタスクの期間を減らす(つまり急いで作る)事になりますし、もしくは実装すべき事柄を削っていきます。

で、もしももとの計画通りバッファ手前で終わりそうなら、バッファ期間でリファクタリングをするなり、次のタスクを始めるなりしましょう。その際はスケジュールのほうも書き換えて前倒しスケジュールにしましょう。

実装レベルまで細かくスケジュールリング

まず、スケジュールリングするために、企画→仕様をある程度(ガチガチにすることは実質不可能だと思います)が詰めておきます。

ある程度詰めてくるとわかるのは『どういう機能が必要か』ということです。当たり前ですが機能は実装しないとならないものです。

例えばプレイヤーが

- 左右に動く
- スピードアップ
- ジャンプする
- 攻撃する
- 死ぬ

などは「機能」ですね。もちろんこの「機能」は「登場人物」ごとに必要なもので

- プレイヤー
- 敵
- ボス

だけでなく

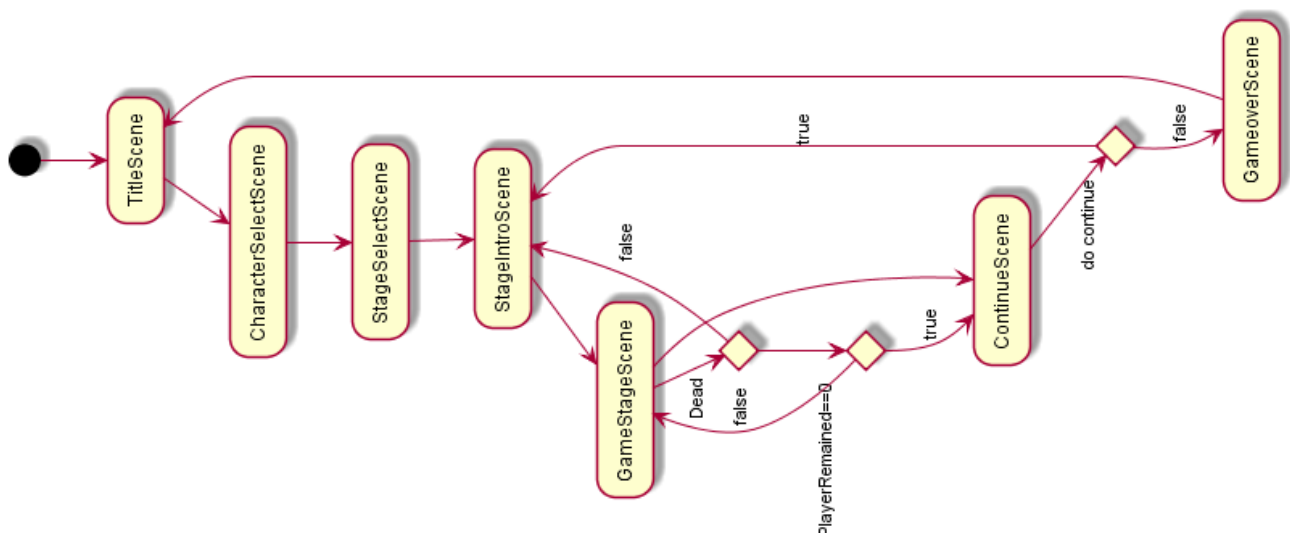
- ゲームシーン
- ゲーム全体

といったところまで考えなければなりません。そうなってくるともう「設計」に近いところまで考える必要が出てきます。まあともかく

「登場人物」と「機能」を思いつく限り紙やテキストファイルに書き出します。「もう漏れはないかな？」というところまで書き出します。

ここができてないと想定外の登場人物や機能が出てきてまた遅れが発生します。

あと、機能だけじゃなくて流れ的なものもありますね。いわゆるフローチャート等で表されるものです。できればフローチャートも手書きでもエクセルでもいいので自分で作っておきましょう。



ちょっと縦長だったので横にしていますが、こんな感じで(PlantUML というツールを使って出力してます。フリーのツールなので PlantUML で検索したらすぐにできますし、VisualStudioCode と組み合わせるともっと楽ちんです)

仕様書を作るときのコツ

仕様書の最初のコツ

※できる限り**仕様の漏れがない**ように意識してください(完璧は無理だけど見落としがあるとならだけ予想しない工数が発生してスケジュールがグチャグチャになるのを防ぐためです)

仕様書にしていく工程

- ① まずは紙でもエクセルの別シートでもよいので必要となる「要素」を「すべて」書き出そう。勿論ここで本当にすべて出てくるわけではないと思いますが「今思いつく限りのすべての要素」を書き出しましょう。
- ② 本当に、本当にもうそのゲームの「要素」をすべて書き出したと思ったら「選定」しましょう。「やっぱこれイラネ」とか「これタブってる」を削除します。
- ③ 今度はその②で残った「要素」の「役割」「動作」「状態」についてドバーツと記述します。これも漏れがないかどうかを意識してください。で、これもいらないものを省きます。
- ④ ①～③までを Excel 等にまとめます。

忘れやすい登場人物(オブジェクト)の例

登場人物というと、プレイヤーやエネミーなどの「役者」ばかり思い当たりますが

- カメラ
- 音響(BGM/SE)
- マップ(レベル)
- シーン

などの裏方さんも忘れないでね。

仕様書とオブジェクト指向設計

で、ここまでで「要素」と「役割」と書きましたが、基本的には実装の段階では「要素」が「クラス」となり、「役割」が「メソッド」という風になっていきます。これが「オブジェクト指向設計」です。

ユーザーの存在を意識してね

※登場人物の話をしてきましたが、ゲームというのはユーザーが主役です。ユーザーをもてなす、楽しませるものです。

ユーザーが何をするのか、それに対してどうリアクションするのか、そのリアクションでユーザーに対してどういう感情を期待するのかを意識してください。

作り手はユーザーの「なぜ」や「目的」にこたえられる(示せる)ようにしてください

例)なぜユーザーはプレイヤーを操作するのか

という疑問なら

理由:分身として操作させることでゲームの世界に入り込んでもらう

実装:ゲームの世界に入り込みやすい工夫を行う

という実装につながります。ここまで意識してください。

以下もユーザを意識してほしいポイントです

- ① ユーザーにどういう目的を与えるのか(なぜその目的なのか)
- ② その目的を達成するのに、ユーザーは何をするのか
- ③ ユーザーのアクション(入力とか選択)にたいして、どういうリアクションを行うのか
- ⑤ そのリアクションでユーザーはどのような感情になるのか

感情→びっくり、こわい、興奮、達成感など

プロトタイプ作り方

あくまでも『プロトタイプ(試作品)』であることを忘れずに。あまり時間をかけないようにしてください。

例えば、ゲームのコア部分が技術的に時間がかかるようなものなら、相当に簡略化して Unity なり UE4 なりでゲームのルールの雰囲気だけでも作ってください。

例えばスプラトゥーンなら、『塗りつぶし』のアルゴリズムがコアですが、それなりの技術が必要で(デカルルやら透視投影やらの理解…)、下手するところにクッソ時間がかかっちゃうわけです。じゃあどうするかというと Unity の Cube なりなんなりでステージを構成して、TPS でも FPS でも球(弾)を発射して、それに当たったブロックの色を変えていって『陣取りゲーム』であることを示したりします。

あと、対 CPU 戦もかなり難しいことになると思いますので、ここではきちんとした AI ではなくランダムに動くとか、雰囲気だけ見せるなら、ランダムに向こうの色が増えていくとか、そういうので簡単に雰囲気を見せるようにしてください。

また脱出ゲームとか一定の課題をこなすゲームの場合『もう一度プレイするモチベーション』が得にくい。ため RTA ではないですが、クリアまでにかかった時間をランキング表示したりします(プロトタイプ版では 1~9 位までタミーで一番下に自分の記録を表示…など)。

要は『ユーザーが遊ぶモチベーションを保ちますよ〜』というのをハリボテでもいいから示しましょう。

ガントチャートの意義と構築について

なんで細かくスケジュール切ってガントチャートを立てるのがそもそも人間というのは遅れます。そして『先延ばし』します。

これはもう『人間の習性』と思ったほうがいいです。『俺は遅れないぞ!』『頑張るぞ!』は無謀です。習性に逆らうからです。思い上がるな小僧が!!ってことです。

まあ何度も言ってるように、直前にならんと頑張らんわけですよ。夏休みの宿題でも、試験対策でも、制作でも、就活でも、直前で慌てて苦しい思いをするわけです。

これはねえ…精神論じゃ無理です。抗えません。

そういうものの助けとして『遅れを視覚化する』ツールがガントチャートなわけです。あと、遅れと『残り作業どれくらいあんの?』というのを視覚化することによって、今の作業の『細かい締め切り』を『たくさん』つくっておきます。

そうすると人間、細かかろうがでかかろうが、『締め切りは守ろうとする』この習性を利用して、『いつまでに何を終わらせるか』を細かく設定して自分を焦らせます。

『今の進行度合い』に関しても、本当は細かく定義してないといくらでもごまかします。『これくらいかなー?』でやると必ずごまかします。これだと絶対終わりませんし、最後に苦しみます。

で、ガントチャート立てるんだったら

①仕様書に書いたことは盛り込んでおく事

ここで抜けてると、作業も抜けるんで、想定外のタスクが発生し遅れる

②タスクの粒度(細かさ)は1日でできる作業くらいを想定してください

1週間の作業が1タスクだと、でかすぎます←遅れます

1週間とるとして、それを1日1日に分解してください。それをチケットとしてあげてください。

③場合によっては1日に複数タスクをこなせると思いますが、それは2タスク

チケット上げて、締め切りを1日にしてください。

④必ず1週間の最後に『マイルストーン』(ここまでにはこの状態に必ずするぞ)

を設定し、それを守ってください。守れない場合は土日消費して遅れを解消してください。

(何度も言うけど、この時期、悲しいけど遊ぶ暇がほとんどないと思ってください)

プログラミング Tips

なお、参考に作ったテストプログラムを

<https://github.com/boxerprogrammer/TestCode>

に上げています。ぼちぼち解説していきますが、なんかの参考になるかもしれません。

うん、もうね、企画とかの話をするの向かないわ、わし。

ということで、プログラミング Tips です。で、もう、わしの専門分野というか、それはシェータ書いたりとかなので、その話をします。

シェータについて

シェータ書く前にさ

言っておくきたい事があるんですが、みんないつも使ってる、DrawGraph とか、DrawRotaGraph とか、DrawRectRotaGraph とか、もうあれ初心者向けすぎるからやめませんか？他所のクラスならいざ知らず、ここ『それなりにできる人たち』のクラスのはずだし、別に全面的に使うのやめろっていうわけじゃないけど、もうメインで使うのやめんか？

と思います。何故かというと、初心者向けな設定にするために色々と非効率だし(前期にも言ったけど DrawCall を減らすという観点からするとマズすぎる)、あれはさ本当に入門用のよね。

君らの提出したプログラムを見る側もあのライブラリの事はたぶん知ってるか、見たらだいたい予想つくのね。で、別に DxLib じたいを使うのはいいんだけど

- いつまで DrawGraph ばかり使っとるん…こいつ仕組みが分かってねえなあ
- 自分でほかの Draw 系調べようと思わんのかね…そんな奴ア…いらん

と、まあこんな感じで『甘えんじゃねえよ』と思われるわけですね。

というわけで、DxLib.h の中身を見てください。で、Shader は 3D にしか使えないと思ってるかもしれませんが、2D の関数もありますからね？

https://dxlib.xsrv.jp/function/dxfunc_3d_shader.html#R17N32

とかを見てください

2D に関しては

DrawPolygon2DToShader

とか

DrawPolygonIndexed2DToShader

DrawPrimitive2DToShader

DrawPrimitiveIndexed2DToShader

とかの関数しか使えません。

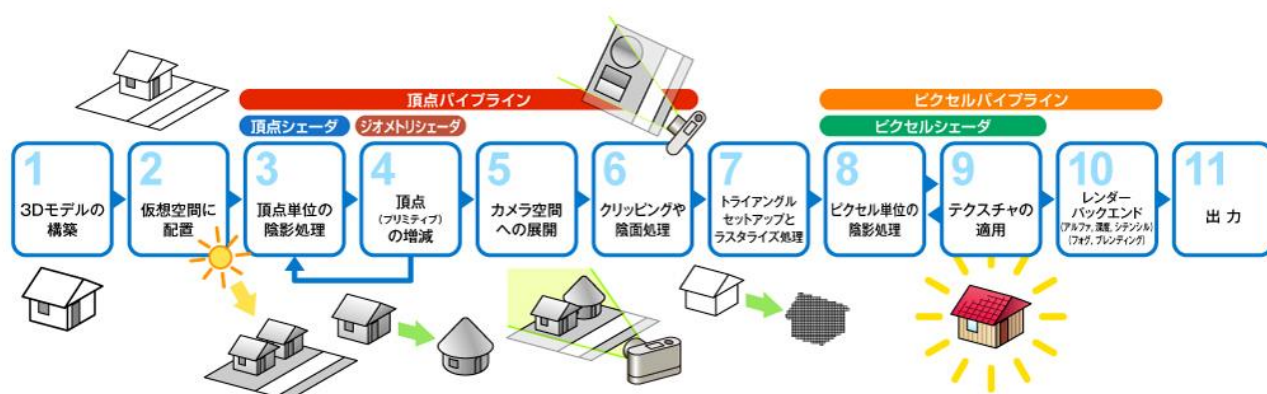
これらの関数はつまるところ、多角形を描画するためのプログラムです。DxLib の 2D ではたいてい長方形

に描画をしていたと思うのですが、これらの関数でそれをやろうとすると『4つの頂点座標を指定』『カラーを指定』『UV 値を指定』しなければなりません。

『なりません』て言いましたが、これらは本来は自前でやるべきもので、全部 DxLib くんが簡単にするために彼の内部でこれらの事をやっていたにすぎないのです。ともかく DxLib くんに感謝するとともに、自分でその部分を書く覚悟を決めてください。

レンダリングパイプラインについて軽く…

2D だろうが 3D だろうが、頂点で構成された物体を画面上に表示するためには以下の流れにのっとっている。



[\(3D グラフィックス・マニアックス\(9\) 3D グラフィックスの概念とレンダリングパイプライン\(1\) | マイナビニュース\(mynavi.jp\)より抜粋\)](#)

別にこの流れを知らなくてももちろんシェータは書けるんだけど、流れを知っておいたほうがいい。圧倒的に良い。

っていうか、コンピュータグラフィックス表現が出てくる仕事に関わるなら大雑把にでも把握しておく必要がある図です。

ただ、まあ基礎知識がないとこの図ですら難しいと感じる人もいるでしょうから大雑把に言うと、まずグラフィックスパイプラインにおいて

『頂点のない表示物は存在しない…ていうか表示対象にならない』

です。皆さんが DxLib::DrawGraph でお手軽に描いている画像も、4つの頂点から作られています。DxLib に慣れ過ぎるとこういう『当たり前の事』すら意識しなくなるので、ちょっと気をつけましょう。

(逆にこの辺は 2D でも頂点が見えてる Unity や UE4 のほうが頂点意識できるかも?)

で、このグラフィックスパイプラインにおけるデータというのは

- 頂点情報(必須)
- テクスチャ(必須ではない)

- シェータ(必須)

ということになります。実はシェータもデータの一つというあつかいです。テクスチャが必須ではないと書きましたが、これは DxLib で言うところの DrawBox や DrawCircle などの、テクスチャが必要ないものと考えてもらうといいかなと思います。

つまり、DrawGraph などでは意識させないようになってますが、実際は

- ① 4 頂点をグラボ(VRAM)に投げる
- ② テクスチャをグラボ(VRAM)に投げる
- ③ 頂点をグラフィックスパイプラインに流し込む
- ④ 頂点シェータで頂点変換する(ピクセル→0.0~1.0)
- ⑤ ラスタライザによってピクセルシェータが呼ばれる
- ⑥ ピクセルシェータ内で頂点の UV に合わせてテクスチャの色をとってくる
- ⑦ 画面に表示

って感じです。めんどくせえでしょ？これやってくれてるんだから、良い子のみんなは、DxLib とその作者さんに感謝をして…生きようね!!

さて、上の時点で大雑把な『レンダリングパイプライン(グラフィックスパイプラインとも)』について書きましたが

流れは

- ① 頂点流し込み
 - ② 頂点シェータで座標変換→(ちょっとややこしいのはピクセル単位の座標、幅、高さをいったん 0~1 までの範囲に直します…で、これは実は最終的に画面に表示される際に『ビューポート』と言って、画面の幅、高さに合われます。『回りくどい!!!ジューラル星人くらい回りくどい!!!』と思われるかもしれませんが、もともとの辺は 3D のための仕組みだからね、しょうがないね)
 - ③ ラスタライザでピクセル化→(頂点の座標データの状況はベクタデータという。これを 1ピクセル 1ピクセルのデータ(ラスタデータ)に変換する事を『ラスタライズ』という)
 - ④ ピクセルシェータで色を塗る→(テクスチャデータは頂点データと違ってレンダリングパイプラインに流し込まれるわけではなく、ピクセルシェータが『VRAM上のメモリ』を参照し、色を決定する)
- という流れです。

勿論、これは『テクスチャの内容をそのまま表示するだけなら』の話です。3D の物体や『法線データ』と絡めていくともっともっとややこしくなっていきます。

まずはこの『流れ』を把握してください

3D の知識の基礎について

まあ、UE4 とか、Unity とか DxLib を使うとお手軽に 3D の実装ができてしまうのですが、基礎知識がないと思いがけない不具合の原因が分からず悩むことになります(深度ファイティングやシャドウアクネなど、そもそも何が起きてるのか分からないとどうしようもない)

割と UE4 とか Unity だとその辺、ものごっつい科学力でいい感じにしてくれてるんだけど、DxLib はそうでもない。

このため、3D の基礎くらいは知っておかんと多少は、ね。ということだ。

3D グラフィックスの事を知りたければ、ワイの本を読むか

<https://news.mynavi.jp/article/graphics-1/>

ここを読むかですね。この連載はガチで分かりやすいし、知っておくべきことばかりだし、読みやすいので、こっから就職内定までは、ソシャゲも漫画もやめてその時間をこの連載を読むのに充ててもらいたい。

それすらできなきゃもう業界就職は諦めて、どうぞ。

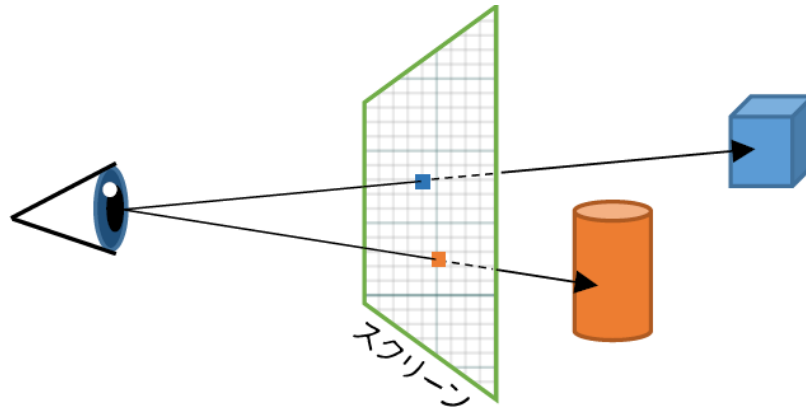
別に理解できなくてもいい、大雑把にどういう世界か知っておけばいい。勿論、超わかりやすい解説なので、理解できるに越したことはないが…

とはいえ、これを読むことをメインにしてもいいけない。基本的に皆さんの就活は『制作物』頼みなのだから。そこら辺のバランスはとって。まあ暇なときにというか、制作してない時間は読むって感じで。

スキャンライン法と射影変換

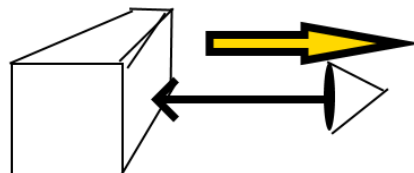
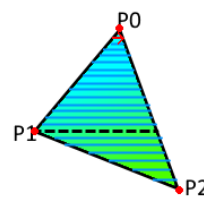
あれ？この辺って CG 概論とかで知ってるもんだと思ってたけど、まあ実践と絡んでなければ身についてないのも仕方ないか…？

以前にレイトレの授業でお話した通り、3D の要素として仮想空間には視点とスクリーンと描画対象オブジェクトがあります。



これがレイトレの仕組みでしたね？これに対してゲーム開発で現状メインで使用されてるレンダリング方式が『スキャンライン法』です。

スキャンライン法
(塗りつぶす前に3D→2D変換が必要)
プロジェクション行列(射影行列)
現在のほとんどのゲームに使われてる
2Dの三角形を構成する3頂点の座標のみがあればいい



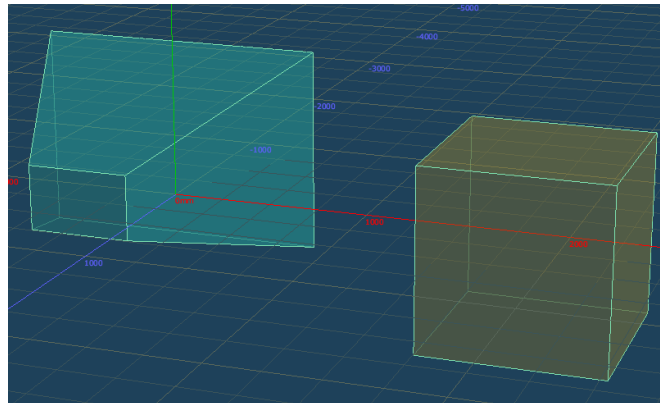
大雑把に言うと『三角形を塗りつぶすためのアルゴリズム』です。で、レンダリングパイプラインの重要な要素『ラスタライザ』というのがあります。これはベクターデータをラスタデータに変換するものです。三角形の場合『ベクターデータ』とは3つの頂点の座標を表します。

これを画面に出力する際にはピクセルの情報に変換しないと画面に出てきません。3頂点から、3角形内部を塗りつぶすピクセルの座標データに変換するのが『ラスタライザ』でピクセル単位になったデータを『ラスタデータ』といいます。

このラスタデータになったピクセル1つ1つに対して『ピクセルシェーダ』が実行されます。

さらに言うと、3D の場合はこれに加えてちょっとややこしくて、ラスタライズするためには、3D のデータのまじゃラスタライズできません。

ラスライザはあくまでも『ただピクセル化』するためのものなので当然 2D 情報です。なので、三角形を塗りつぶすにしても P0,P1,P2 は 2D になってる必要があります。ということは 3D の頂点情報を 2D の頂点情報に変換する必要があります。3D でも最終的に 2D のディスプレイに表示される以上は 2D のデータにする必要があるというわけですね。



で、遠近法がかかっていると↑の画像の左側の図形のように視点から広がる形をしているので、3D の台形みたいな形をしています。これを『視錐台』といいます。これを 2D として使えるようにするために右のような長方形の形にします。

それをやる変換を『射影変換』『プロジェクション変換』『パースペクティブ変換』といいます。で、これは 4x4 行列なので射影行列、プロジェクション行列、パースペクティブ行列なんて言ったりします。この行列演算を行うのが『頂点シェーダ』です

レンダリングパイプライン自体は、これ頂点の流れに過ぎないのです。

頂点→3D⇒2D 変換→ラスライズ→ピクセルシェーダ→レンダーターゲット
の流れですね。

『あれ？2D 情報言うたけど、直方体やから 3D ちゃうん？』と思った人はもう少しお待ちください。ともかくまあそういう変換がかかって 2D 用の直方体になってると思ってください。

で、レンダリングパイプラインを思い出してほしいのですが、先頭で流し込まれるのは『頂点情報のみ』です。

じゃあ、テクスチャデータはというと実はあらかじめ『グラボ上に存在する』というイメージです。

イメージ的には LoadGraph した時点でグラボ上のメモリに配置されています。DrawGraph の時点でグラボに飛んでいくと思ってるかもしれませんがそうではなく、描画時にはすでにグラボ上に乗っかってます。このイメージは軽くもっておいてください。

で、Draw したときに、ピクセルシェーダ上でグラボメモリ上のテクスチャを探して、そのテクスチャデータをもとにポリゴン描画の際にテクスチャの画素値をとってきて、その値を使用しているわけです。

で、また先ほどの台形の話に戻りますが、3Dにおいて『見える範囲』というのは、3Dの台形みたいな形をしています。このことを『視錐台』もしくは『クリッピングボリューム』といいます。要は『この範囲の内部しか見えません』というものです。

ここで重要なのが Near と Far の概念で、視点に近いほうを Near で遠いほうを Far といいます。左右の見える範囲だけではなく、手前遠く of 見える範囲も規定しているんですね。最終的に $-1 \sim 1$ の直方体に変換する以上、奥が無限大でもいいけないし、手前が 0 でもまずいわけです。

で、この台形の形がどんな形をしていても最終的には(ラスター化前には) $X(-1 \sim 1), Y(-1 \sim 1), Z(0 \sim 1)$ の直方体に変換されます。

【こういう範囲を無理やり $0 \sim 1$ とか $-1 \sim 1$ に変換する事を正規化といいます】で、こういう仕組み上『あまり Near と Far の範囲が広い』と問題が発生します。

例えば $1 \sim 100$ を $0 \sim 1$ に変換するのと、 $0.1 \sim 1000000$ を $0 \sim 1$ に変換するのじゃ、量子化レベルが違うのは分かりますね？つまり奥行き方向の情報量は一定なのだから、あまり広げると奥行きの解像度が落ちる→結果的に『Z(深度)ファイティング』が発生します。

Z ファイティングというのは何かというと、そもそも描画時に手前億判定をするために深度バッファ法というので実は隠面消去をしています。ようは手前の物体に隠された奥にある物体は描画しないようにしてるわけですね。

で、例えば両手を前後に配置して、手前の手で奥の手が隠されるような事をするのが深度バッファ法ですが、このバッファの解像度が下がると手前の手と奥の手が同じ深度値になってしまいます。この結果どちらが手前に来たらいいのかわからない。24bit もしくは 32bit の範囲で手前が行ったり来たりしてちらつく→深度ファイティングといいます。

2D のシェーダから挑戦しよう

さて、いいよシェーダを書きたいと思うんですが、シェーダはコンパイルしないと使えません。え？スクリプトみたいなものでしょ？って思われるかもしれませんが、れっきとしたコンパイラがあります。

Direct3D を使う時なんかはロード時にコンパイルするために `d3dcompiler.lib` なんかをリンクしてロード時にコンパイルしたりするんですが、DxLib では事前にコンパイルすることになります。

DxLib ではシェーダコンパイラとして `fxc.exe` を使用しますので、もしかしたら(`dxcc.exe` が OK かも、その場合は GitHub から落としてください)、`fxc.exe` がどこにもないかもしれません。

で、ネットを探すとありますが、この時注意しないと、がつりウィルスとかマルウェアとか食われます。というのがあって、一応、Teams のいつものところに `fxc.exe` を置いています。

ので、試したい人は、こちらを使ってください。

使い方は

```
fxc.exe /T ps_4_0_level_9_1 /O3 /Fc testps.psh /Fo testps.pso testps.fx
```

てな感じで使います。ここでは自分で書くのは `testps.fx` のみです。

とはいえ、2D のシェーダ例なんてあんまりネットにも転がってないので、まずは簡単なものから挑戦しましょう。

DxLib におけるお勧めの「初歩シェーダ習得までの流れ」

まずは PixelShader だけいじって、画像を白黒化してみよう。

①例えば `rgb` を $0.299r + 0.587g + 0.144b$

を輝度 Y として、 $r=g=b=Y$;

これでモノクロ画像になりますとても簡単なので、まずはロードしたグラフィックをモノクロにするところから試してみましょう。

もししんどい場合はやらなくて結構です。

②それができたら、色を反転させてみましょう。1-カラーでできます。

DrawGraph の代わりを作る

ちなみに手っ取り早くやるなら、こういう関数を作っときます。

```
void MyDrawGraph(int x, int y, int width, int height) {  
    array<VERTEX2DSHADER, 4> verts;  
    for (auto& v : verts) {  
        v.rhw = 1.0f;
```

```

        v.dif = GetColorU8(0xff, 0xff, 0xff, 0xff);
        v.u = 0.0f;
        v.v = 0.0f;
        v.su = 0.0f;
        v.sv = 0.0f;
        v.pos.z = 0.0f;
        v.spc = GetColorU8(0, 0, 0, 0);
    }
    verts[0].pos.x = x;
    verts[0].pos.y = y;
    verts[1].pos.x = x+width;
    verts[1].pos.y = y;
    verts[1].u = 1.0f;
    verts[2].pos.x = x;
    verts[2].pos.y = y+height;
    verts[2].v = 1.0f;
    verts[3].pos.x = x + width;
    verts[3].pos.y = y + height;
    verts[3].u = 1.0f;
    verts[3].v = 1.0f;
    DrawPrimitive2DToShader(verts.data(), verts.size(), DX_PRIMTYPE_TRIANGLESTRIP);
}

```

見りゃ分かると思いますが、この関数ではグラフィクスハンドルを指定していません。前にも言いましたが DrawGraph とは仕組みが違います。どうしても DrawGraph を再現するなら。

```

void MyDrawGraph(int texH, int normH, int shaderH, int x, int y, int width, int height) {
    SetUseTextureToShader (0, texH);
    SetUseTextureToShader (1, normH);
    SetUsePixelShader (shaderH);
    MyDrawGraph(x, y, width, height);
}

```

```

void MyDrawGraph(int x, int y, int graphH, int shaderH) {
    SetUseTextureToShader (0, graphH);
    SetUsePixelShader (shaderH);
    int w, h;
    GetGraphSize(graphH, &w, &h);
    MyDrawGraph(graphH, -1, shaderH, x, y, w, h);
}

```


こんな感じでしょうが、僕はあまりお勧めはしません。せいぜい4点の指定を棄にするくらいでいいでしょう。

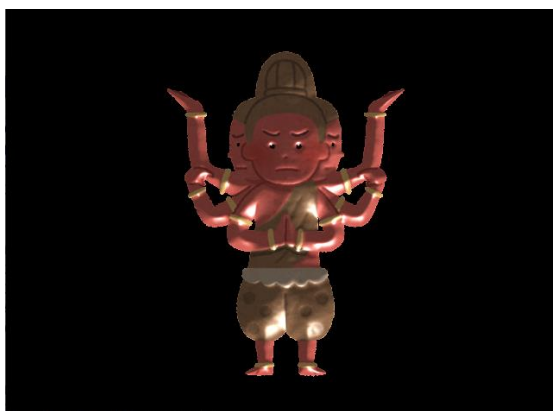
ちなみにシェータを使用するにはシェータの読み込みが必要です。

```
auto pshandle = LoadPixelShader(L"testps.pso");
```

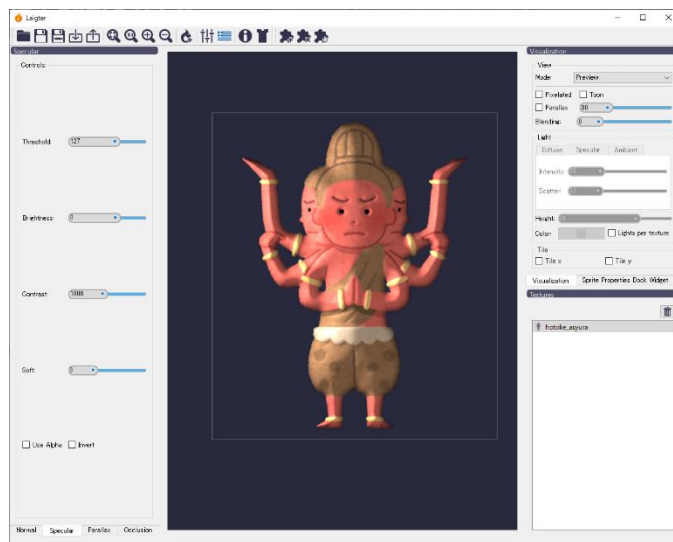
コンパイル後のシェータしか使えませんので注意してください。

2Dで3Dっぽくする(法線マップによるシェーディング)

はい、お待ちかねじゃないですかね。これ。



これの実装について話します。Laigter の使い方については別に解説しませんので、各自で使いこなしてください。



即席で作ったものなので Laigter と見た目は異なりますが、とにかく3Dのように見えてるのが分かるかなと思います。

Laigter を使うと、2D 絵に合わせた法線マップを作ってくれます。



はい、これがあればもう3Dっぽくできます。

法線マップについて

↑の図の右側の青紫のやつが法線マップです。

ノーマルマップとは…(目的)そもそもノーマルマップは物体表面に凸凹を貼り付けて、ぺたんこの表面に凸凹があるように見せかける。

ノーマルマップ画像の話

ノーマルとは法線の事です。法線ベクトルを画像化し、物体表示時にその画像をもとに凸凹を付けることで物体表面に凸凹がつくで、ここで問題があります。

法線ベクトル $N(x,y,z)$ の範囲は $(-1\sim1, -1\sim1, -1\sim1)$

ただし正規化されてるんで、 $-1\sim1$ を超えることはない。それに対して「画像」の場合、RGB の「色」の集合体色 (R,G,B) の取りうる範囲は $(0\sim1, 0\sim1, 0\sim1)$ のように法線ベクトルとは違う。

じゃあどうすんのかというと、こうやって保存されてます

$$r=(x+1)/2$$

このように保存されている

$$g=(-y-1)/2$$

$$b=(z+1)/2$$

じゃあ、この rgb から xyz を復元するには当然ですが

$$x=(r-0.5)*2.0$$

$$y=(0.5-g)*2.0$$

$$z=(b-0.5)*2.0$$

という計算式になります。

$$x=2.0r-1.0$$

はい、ここに注意してプログラムを書いてください。

実装

DxLib ではテクスチャを 2 枚以上載せられますので、2 枚目を法線マップとして扱います。

法線マップも普通の画像としてロードして可です。そのうえで

```
SetUseTextureToShader (0, graphH);
```

```
SetUseTextureToShader (1, normH);
```

のようにセットします。番号が異なるのがポイントで、それぞれシェータ側では番号で区別します。

```
Texture2D tex: register( t0 );
```

```
Texture2D norm: register( t1 );
```

はい、この辺のルールは DirectX12 とかと同じです。+ なんとか、の後ろの番号が対応しているわけです。

で、ノーマルマップ情報が前述のとおり、rgb→xyz 変換の必要があるので

```
float4 n=norm.Sample(smp,PSInput.uv);
```

```
n.xyz=n.xyz*2-1;
```

```
n.y=-n.y;
```

とします。

で、

```
float b=saturate(saturate(dot(normalize(float3(1,0,0)),n.xyz)+0.25));
```

で、こんな感じで書くと右側からライトが当たってるような感じで輝度計算されます。

この輝度値をもとのテクスチャに乗算してあげればいいわけです。

なお、saturate 関数というのは clamp(value,0.0,1.0)と同義ですが、hlsl においてはノーコストで使えるので覚えておきましょう。

ライトレやってる皆さんにはいちいち細かく解説しませんが、単純なランバート余弦則で明るさを決定しています。

先ほどの画像はさらに調子に乗って、スペキュラも入れてみて

```
saturate(normalize(reflect(float3(-1,0,0),n.xyz)).z)
```

をスペキュラ値にしてるだけです。あ、この結果をさらに n 乗しますよ？スペキュラは n 乗しないと明るい範囲が広がりますから。

```
float s=pow(  
    saturate(normalize(  
        reflect(  
            lightVec  
            ,n.xyz  
        )  
    ).z),10.0);
```

ちなみに z 値を使っているのは、2D なので、視線方向が z 方向固定だからです。

ちょっとややこしい(マニュアルが嘘ついてる)定数バッファ

例えば↑のライティングは右側から固定なので、ライトを回転させたりしたいとすると『定数バッファ』を使


って、CPU 側の情報を GPU に流してあげる必要があります。
その場合に使用されるのが定数バッファなのですが…。

ちなみに DxLib で定数バッファを使おうとしてマニュアルを見ると『PSSetConstF を使え』みたいに書いてるんだよね。

https://dxlib.xsrv.jp/function/dxfunc_3d_shader.html#R17N17

でも見落としちゃいけない文言が…

「（この関数は Direct3D 9 用の関数です、Direct3D 11 では効果がありませんので注意してください）」

…は？ 

ムカつきますねえ。そしてその DirectX11 でのやり方は一切書いてない。ちなみにたぶん初心者は9とか11とか分からずにやるだろうから、PSConstF を使いまくって失敗して訳が分からず泣きを見ます。注意しましょう。

で、ちょっとクツソややこしいですが、定数バッファを作ります。この辺もう DxLib というより Dx11 なのでツラいっすね。

手順

- ① 定数バッファを作る
- ② 定数バッファのアドレスを取得する
- ③ アドレスをキャストして中身を書き換える
- ④ 書き換えたことを通知する

が必要になります。なお、メイン③④はループ内で行うことになります。

まず定数バッファの確保

```
auto cbuffer=CreateShaderConstantBuffer(sizeof(float)*4);
```

はい、これ4つ確保して無駄じゃん!と思うかもしれませんが、アライメントが16byteなので、こうしないとバグります(これもマニュアルにも何も書いてない、ぶっ飛ばすぞ)

で、今回は float1 つを『角度』としてしか使用しないため以下のようにして受け取ります。

```
float* gangle = static_cast<float*>(GetBufferShaderConstantBuffer(cbuffer));
```

はい、この gangle が定数バッファのアドレスを示しています。なのでここを書き換えます。

```
gangle[0] = angle;
```

はい、まあ書き換え方は各自にお任せします。

で、書き換えた後は

```
UpdateShaderConstantBuffer(cbuffer);
```

```
SetShaderConstantBuffer(cbuffer, DX_SHADERTYPE_PIXEL, 0);
```

を呼び出します。Update は GPU 側のメモリ書き換えを完了し、Set なんとかは 0 番定数にバッファの内容を割り当てます。

あとはシェータ側ですが

```
float angle : register( c0 );
```

のように書けば、内容が angle に入っています。

これを利用してライト方向を

```
float3(cos(angle),sin(angle),0))
```

なんてやれば、光を回転させたりできます。以上。

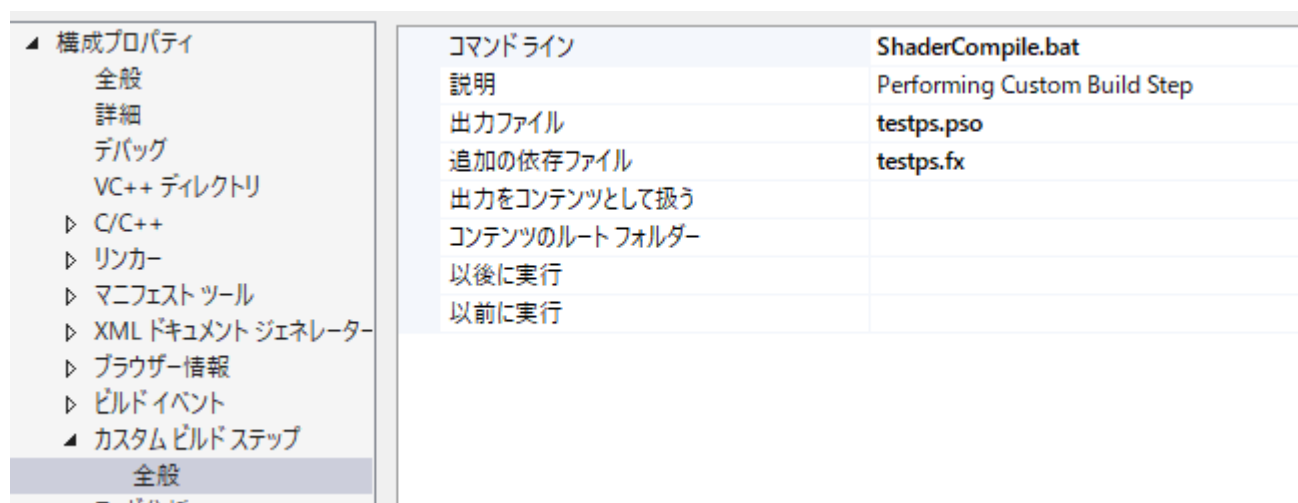
いちいちシェータコンパイルを手動でやるのが面倒だから自動でやる
もともと

```
fxc.exe /T ps_5_0 /O3 /E "main" /Fo testps.pso testps.fx
```

てな感じでシェータコンパイルを書いてたと思いますが、正直これを毎度実行しては面倒です。
ということでまずこの内容をバッチに書きます。バッチの作り方は分かりますね？

で、次に、VS のビルド時に同時にコンパイルされるようにしてみます。

プロジェクトプロパティ→カスタムビルドステップ



コマンドライン	ShaderCompile.bat
説明	Performing Custom Build Step
出力ファイル	testps.pso
追加の依存ファイル	testps.fx
出力をコンテンツとして扱う	
コンテンツのルート フォルダー	
以後に実行	
以前に実行	

はい、ここでコマンドラインに先ほどのバッチを書いておきます。で、ここが重要なのですが、『出力ファイル』と『依存ファイル』です。

これは『依存ファイル』に変更があった(つまり出力ファイルよりも更新日時が新しい)場合はコマンドラインを実行し、そうでなかったら実行しない→つまり無駄なコンパイルを行わない。

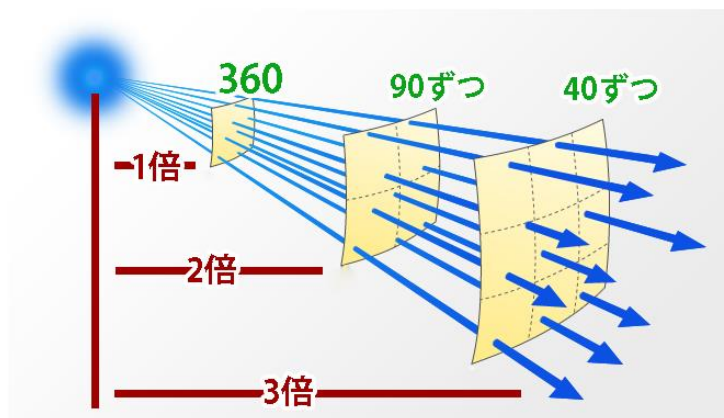
です。これである程度らくになりますね。

点光源にしてみる

さっきまでの話はあくまでも『平行光線』の話でした。

では点光源の場合は何が違うのでしょうか？それはまず点光源なのでピクセルの座標ごとに光源へのベクトルが変わります。

次に点光源は逆 2 乗の法則で減衰します。



初めて聞く人には難しいかもしれませんが、とにかく『距離の 2 乗に反比例して光が弱くなる』です

$$I_{out} = I_{in} \frac{k}{d^2}$$

こんな感じですね。

はい、では点光源に必要なものは座標と『k』…これ点光源の強さってことでいいのかな。Unity やら UE4 だと Intensity ってやつにあたりますが、ピクセル計算でやる場合はこれ 10000 くらいしないと全然見えません。

これをまとめて float4 で渡します。

light.xyz と light.w

です。

```
float4 light : register( c0 );
```

はい、こうやって 4 つ分渡してきます。で、ライトベクトルを作ります。xyz と差分とってますが xy だけでもいいでしょう。どうせ 2D だと z 意味ないですし

```
float3 lightVec=light.xyz-PSInput.pos.xyz;
```

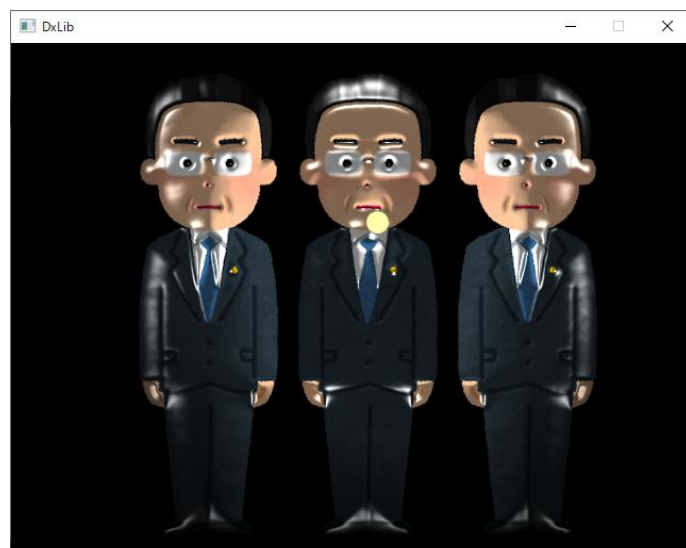
で、逆 2 乗なので

```
float intensity=saturate(light.w/dot(lightVec,lightVec));
```

とします。

で、あとは前と同様に、ディフューズとスペキュラを計算して、最後にこの intensity を書けます。

```
PSOutput.Output.rgb*=intensity;
```



おわり。

お手軽に画面の賑やかしに使える『レイマーチング』

先日、twitter で面白いのを見かけました。



おおー、こういうので使えるかー。これは面白い。2D のレイマーチングも使えるなあ。アイデアだなあ。って思いました。

https://twitter.com/yagiri_pg/status/1449659771649343494

で、2D レイマーチングを UI に使ってる例だと

<https://blog.sb1.io/intro-to-2d-signed-distance-functions/>

なんてのもありました。

まあ去年の話からぶっこ抜いてくるわけですが

レイマーチング基本というか初歩の話

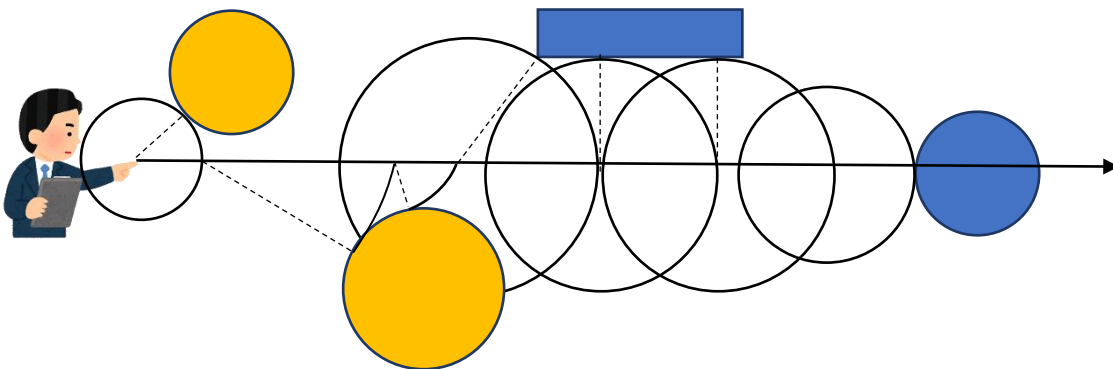
レイトレーシングは衝突点を求めるところから話が始まっていて、球体と直線の交点を求めるのが出発点でした。

レイマーチングはその出発点が少し違って、SDF(Signed Distance Function)つまり距離関数(符号付き)を用いて物体表面を記述するものです。

例えば球体との距離であれば

```
float SDFSphere(float3 pos, float3 center, float r){  
    return length(center-pos)-r;  
}
```

よくレイマーチングの模式図として



こういう図が用いられますが、いきなりこれを見せられても何が何だか分かりませんね？後から振り返って見れば、適切な図であったことは分かるのですが、初心者に対しては『ナンジャラホイ』って印象を持たれてしまうんじゃないかなあ…って思います。

ひとまずこいつは『数学』からのアプローチではなく『アルゴリズム的アプローチ』で考えたほうがよさそうです。

- ① 視点と視線ベクトルを取得する
- ② 始点=視点とする
- ③ 始点から最も近いオブジェクトとの距離を求める(総当たり)
- ④ ③で得られた距離の分だけ現在の視点から視線ベクトル方向に進んだ所を始点とする
- ⑤ ③に戻る

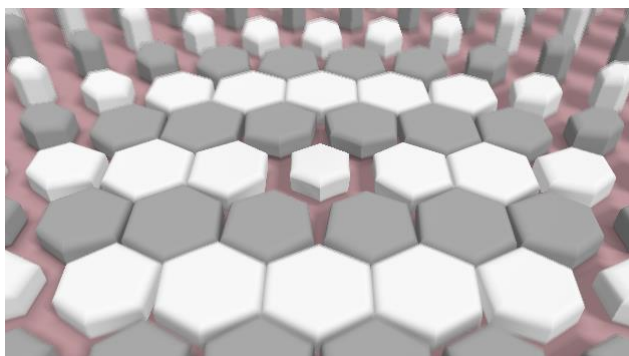
これを数学的に書くと

P_0 = 視点, V = 視線ベクトル(正規化済)

$P_{n+1} = P_n + V * SDF(P_n, \text{Object}_{\min})$

てな感じになると思います。もう何を言ってるんだか分からないと思います。ちなみに SDF は特定のオブジェクトとの最短距離です。

ともかく、一番近いところの最短距離を測って進んでいくことは分かりましたかね？ところでレイマーチングとか言うと以下のような画像を思い浮かべると思います。



そう、幾何学的、そして繰り返し…

先ほどのアルゴリズムでは結局は交点までいかにしてたどり着くのか…という事にしか触れていませんね？

さてさて…このアルゴリズムとこの結果は何の関係があるんでしょうか？それは距離関数を使っているところに秘密があります。

2D における距離関数

例えば、`SDFCircle2D` を以下のように定義します。中心からの距離によってその値を返します。

```
float SDFCircle2D(float2 xy, float2 center, float r) {  
    return length(center - xy) - r;  
}
```

```
float sdf = SDFCircle2D(input.npos.xy, float2(0, 0), r);  
if (sdf < 0) {  
    return float4(1, 1, 0, 1);  
}  
else {  
    return float4(0.5, 0.5, 1, 1);  
}
```

と書くと

となりますが、円の大きさが横広ですので、アスペクト比を乗算しましょう。

```
float2 aspect = float2(w / h, 1);  
float sdf = SDFCircle2D(input.npos.xy*aspect, float2(0, 0), r);
```

fmod で大量生産

何かというと `fmod` を使用します。

fmod 関数は浮動小数点数において、整数型における%演算子のような働きをします。つまり『割った余り』を返すのですが、これがミソです。fmod なので浮動小数点数で割った余りを返します。

0.14 を 0.1 で割った余りは 0.04 という風になりますが、これを利用することで、繰り返し表現を実現することができます。

例えば uv 値に対して 0.1 を fmod してやれば 0.1 の余りが出ます。これにより

```
sdf = length(fmod(input.uv, 0.1))-0.1f;
```

```
if (sdf < 0) {  
    return float4(1, 0.5, 0, 1);  
}else {  
    return float4(0.5, 0.5, 1, 1);  
}
```

何故なのかはわかりますね？これはまず範囲を uv 両方向 0~0.1 になり、そこから 0.1 を引くことで -0.1~0.0 となります。length は距離を測るので三平方状態になり、このように左上中心の円ができます。但しよく見ればわかるようにアスペクト比が反映されてないですね。

```
float2 aspect = float2(w / h, 1);
```

```
sdf = length(fmod(input.uv*aspect, 0.1))-0.1f;
```

とすることによってアスペクト比が調整されます。あとは中心を真ん中に持ってくるだけです。

```
float SDFLatticeCircle2D(float2 xy, float divider) {  
    return length(fmod(xy, divider) - divider / 2) - divider / 2;  
}
```

という関数を作ります。この中にアスペクト比を考慮した uv 値を入れてやれば

ここまでやって思ったことは『一般にレイマーチングって言われてる奴って、距離関数は使うけど本当の意味でのレイマーチングやってなくね？』ってことです。

だって

①視点と視線ベクトルを取得する

②始点=視点とする

③始点から最も近いオブジェクトとの距離を求める(総当たり)

④③で得られた距離の分だけ現在の視点から視線ベクトル方向に進んだ所を始点とする

⑤③に戻る

って言ってるけど、これ総当たりじゃないじゃん…総当たりするまでもなくってというか、どれに当たるか最初

っから決まってるよね？

ってこと。ひとまずは 2D に関してはそう思えますよね。ひとまずはそう思っておきましょう。次にこれを疑似的に立体化したいと思います。まずは疑似的に深さ情報を作ります。

```
float sdf = SDFLattice2D(uv*aspect, divider);  
if ( sdf<0 ) {  
    sdf = abs(sdf)*(1.0/divider);//一※  
    return float4(sdf,sdf,sdf,1);  
};
```

これがどんな結果になるか分かりますか？sdf は※マークの行により 0.0~1.0 となり、中心が 1.0 で周囲



が 0.0 になります。この結果を `return float4(sdf,sdf,sdf,1);` で出力すると

このようになります。まあ実際深度値は手前のほうが小さいのでこれは少し違うんですけどね。それではここから無理やり法線ベクトルを算出し、光を当ててみましょう。

```
sdf = abs(sdf)*(1.0 / divider);  
float2 xy = (fmod(uv*aspect, divider) - (divider / 2))*(1.0 / divider)*float2(1,-1);  
float3 sdfnormal = normalize(float3(xy, -sdf));
```

前述のように sdf は反対側に向けています。で uv 値を SDF 関数と同様の変換を行い、nx,ny を計算、そこから法線ベクトルを作ります。そしてその法線ベクトルとライトベクトルの内積を

```
sdf = max(saturate(dot(normalize(float3(-1, 1, -1)), sdfnormal)), 0.15f);
```

```
return float4(float3(sdf, sdf*0.8,sdf*0.8), 1);
```

のようにとれば…



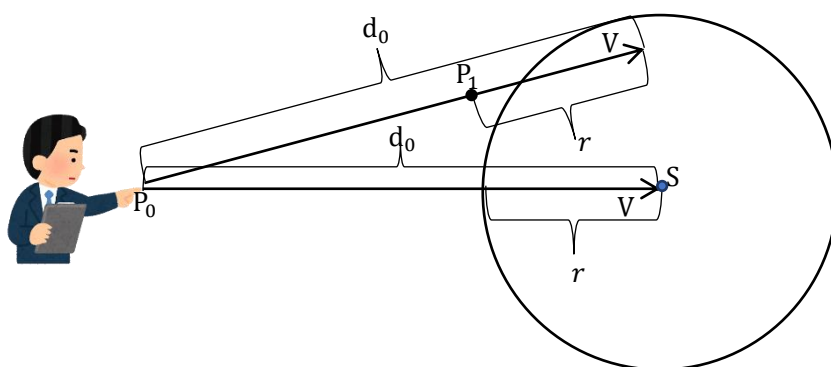
ほら、こんなに簡単にたくさん球体が、こんなん絶当たりできないでしょ？

3D における距離関数とレイマーチング

3D になると途端にレイマーチングになります。でも距離関数が重要なのは変わりません。ただかなり面倒です。

普通にレイトレーシングやらなきゃいけません。まずは再び1つの球体を『レイマーチング』で表示します。レイマーチングの場合は『固定ループ』を用います。ただこれをやると…重い…ループ回数に注意。

まずは1つの球体について考えます。球体の真正面を向いていればきれいにヒットするのですが…上下左右にある程度ズレると1回ではヒットしなくなります。



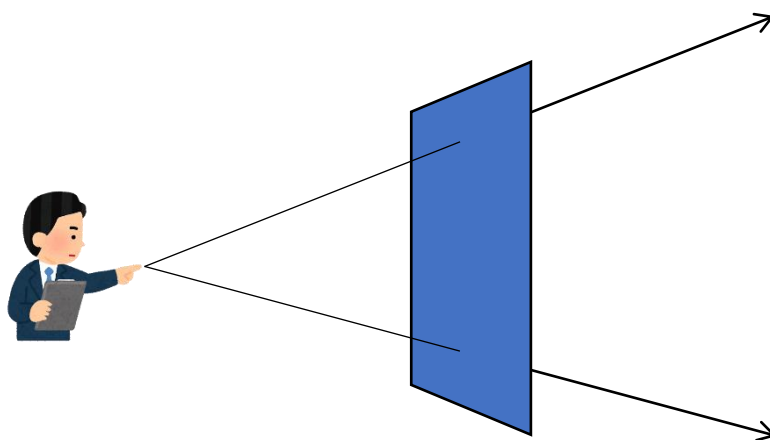
球体の距離関数は $\text{length}() - r$ です。↑の図で言うと

$$d_n = S - p_n$$

$$P_{n+1} = P_n + \hat{V}(d_n - r)$$

こんな感じですかね。これを繰り返して、 $d_n < \epsilon$ になったら処理を打ち切る…と。ちなみにイプシロンは十分に小さな値ね。と言う訳で真正面は1回で処理が終わると言う訳。

さて、レイトレーシングであるからには『スクリーン』の概念と視点の概念が必要なのでこれも用意します。とはいえ、数学の時間のレイトレと違うのはピクセルシェータに入った時点で、スクリーンの場所は決まっ



ているという事。あとは視点を仮にでも用意して視線ベクトルを作り、そこからレイマーチングを行えばいいのだ。とはいえ uv とやっちゃうといろいろとアレなので、pos でやります。pos 言うても SV_POSITION の場合保管されてないので、なければ POSITION の変数作ってスクリーンが1~1 の体でやります。視点は

z=-1 のど真ん中という感じでいいでしょう。つまり

```
float3 eye = float3(0, 0, -2.5); // 視点
float3 tpos = float3(input.tpos.xy*aspect, 0);
float3 ray = normalize(tpos - eye); // レイベクトル(このシェータ内では一度計算したら固定)
float rsph = 1.0f; // 球体の半径
for (int i = 0; i < 64; ++i) {
    float len=SDFSphere3D(eye, rsph);
    eye += ray * len;
    if (len < 0.001f) {
        return float4((float)(64-i)/64.0f, (float)(64-i) / 64.0f, (float)(64-i) / 64.0f, 1);
    }
}
return float4(1, 0, 0, 1);
```

みたいに考えましょう。あ、SDFSphere3D は自分で考えてみてね。

```
float SDFSphere3D(float3 pos, float r) {
    return length(pos - float3(0, 0, 5))-r;
}
```

0,0,5 は仮の球体の中心です。



一応陰影は到達回数によってつけています。もちろん正しい陰ではありませんが球をレンダリングしているのが重要なのです。さて、これをあとは fmod で繰り返してみましょう。

```
float SDFSphereLattice3D(float3 pos, float divider, float r) {
    return length(fmod(pos, divider) - divider / 2) - r;
}
```

としておき、呼び出し側は

```
float len = SDFSphereLattice3D(abs(eye), rsph*2, rsph/2);
```

とでもしてやれば



このように、ザ・レイマーチングってかんじになります。

球体以外の距離関数の参考サイトは

<http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm>

ですのでいろいろ形状を試してみても遊んでみて下さい。

法線ベクトルを返してみる

今の戻り値は距離関数の距離ですので、法線は別の方式で返してあげる必要があります。hlsl で C++ の参照みたいなことをしたい場合はどうするのか今更ですがやってみましょう。

やり方は簡単です。

引数の型名の前に out をつけて関数宣言を行います。それだけです。入力と出力なら inout にします。今回は出力のみなので out としましょう。それで球の距離関数を定義すると…

```
float SDFModSphere3D(float3 xyz, float divider, float r, out float3 normal)
```

こんな感じになります。

で、例えばこの関数の中で normal に法線を返すようにすると

```
normal = normalize(fmod(abs(xyz), divider) - divider / 2)*float3(1, 1, -1);
```




このようになります。あ、ランバートは自分で計算してください。法線に `abs` つけてるんで、ちょっとおかしいことになってますが、そこは各自で調整お願いします。あとこれ、`fmod` が `glsl` と挙動が違うためらしいので、`mod` を自分で実装します。

```
float3 mod(float3 x, float divider){
    return x-y*floor(x/y);
}
```

これを `fmod` の代わりに使いましょう。

球体以外の時の法線

球体以外のときの法線ですが…非常に難しいです。イマノボクニハリカイデキナイ…。

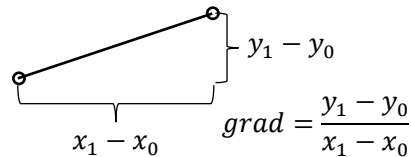
<http://iquilezles.org/www/articles/normalsSDF/normalsSDF.htm>

いちおう日本語の説明の SlideShare がありましたのでご紹介いたします。

<https://www.slideshare.net/shohosoda9/three.js-58238484>

この中にですね『勾配から法線を求める』というのが出てきます。そして『勾配』ってなんやねん!!!となるかなと思います。

勾配ってのは簡単に言うと『傾き』です。傾きと考えると法線を出すのはそれほど難しくないと思いませんか？例えば `xy` の 2 次元しかない時を考えてみましょう。



変化率ともいう事が出来ますね？この法線を作りたかったらどうしたらいいでしょうか？まず考えられるのは $(y_1 - y_0, x_0 - x_1)$ ですね。ひっくり返して片方マイナスにしたらそれが法線になります。

ただしこの考え方では3Dの時にうまくいきませんね。そもそも2Dの時は $y=ax+b$ という表記なのに、3Dの時は法線ベクトルを使ってという流れがなんか不自然ですよ。

まあだから厳密に言うと傾きと勾配は違うっちゃ違うんですよ。どういう違いかというと

$y=f(x)$ という表記は「陽関数」

$f(x,y)=0$ という表記は「陰関数」

と言います。陽関数の代表的なものはそれこそ $y = ax + b$ ですし、陰関数の代表的なものは $x^2 + y^2 = r^2$ です。ここまではよろしいですか？

さて、それでは陰関数 $x^2 + y^2 = 1$ の勾配を求めてみましょう。そして法線ベクトルを考えてみましょう。ここでちょっとだけ「偏微分」の説明をします。ああ、怖くないよ〜。

偏微分ってのはこの場合、 x 方向の偏微分を考える際に y 方向は定数として扱い、 y 方向の偏微分を考える

際には x 方向は定数として扱います。 $f(x,y)=1$ の関数ですが偏微分のそれぞれの方向に関しては $f_x = \frac{\partial f}{\partial x}$ と

し、 $f_y = \frac{\partial f}{\partial y}$ として考えます。さて、そうすると…

$$f_x = \frac{\partial f}{\partial x} = \frac{d(x^2 + y^2 - r^2)}{dx} = 2x$$

$$f_y = \frac{\partial f}{\partial y} = \frac{d(x^2 + y^2 - r^2)}{dy} = 2y$$

ですね？死にそうになってる人はもう少しだけ頑張りましょう。この偏微分が「勾配」です。

この円周上のとある点 $(1,0)$ における法線を考えてみましょう。そしたら $(2x, 2y)$ の $x=1, y=0$ になります。つまり $(2,0)$ になります。正規化すると $(1,0)$ というわけです。結局、任意の面における法線ベクトルというのは、各方向で偏微分してその点の座標を代入すればいいという事になります。

さて、ここで疑問点ですが、なぜ距離関数が偏微分になっているのかという事ですが、ここで偏微分を

$$\lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

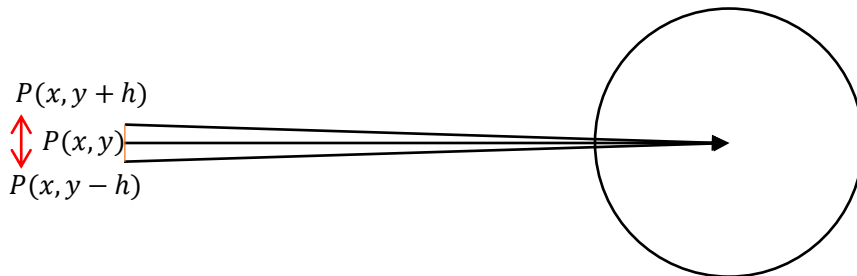
ではなく

$$\lim_{h \rightarrow 0} \frac{f(x+h) - f(x-h)}{2h}$$

として考えています。

※<http://iquilezles.org/www/articles/normalsSDF/normalsSDF.htm>より

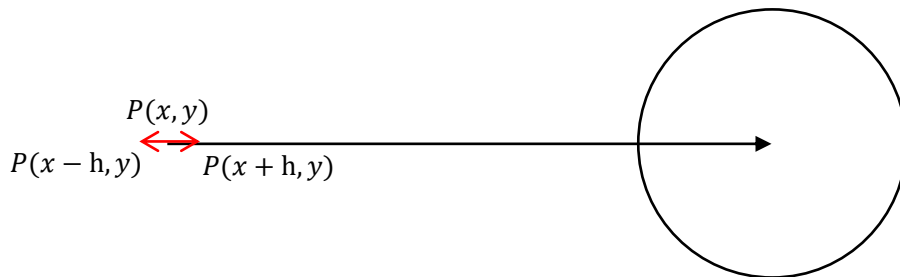
さて、こう考えることでなぜ偏微分になるのかということ、ここは図を描いたほうが分かりやすいです。真正面から見てる時を考えます。



上の場合は $P(x, y+h)$ と $P(x, y-h)$ と球体との距離関数は等しくなってしまいます。そうすると

$$\lim_{h \rightarrow 0} \frac{SDF(x, y+h) - SDF(x, y-h)}{2h} = 0$$

となります。ところがこれが真正面方向になると、話が変わってきます。



$$\lim_{h \rightarrow 0} \frac{SDF(x+h, y) - SDF(x-h, y)}{2h} = \frac{-2h}{2h} = -1$$

となり、それぞれの距離関数の結果をベクトルとすると法線ベクトルになることがわかります。

まあ良く分からんかもしれないので、サンプルコード書いておきます。

```
float2 aspect = float2(float(w) / float(h), 1.0);
float3 start = float3(0, 0, -2.5);
float3 target = float3((input.uv * float2(2, -2) + float2(-1, 1))*aspect, 0);
float3 center = float3(0, 0, 5);
float3 ray = normalize(target - start);
const int trycount = 64;
for (int i = 0; i < trycount; ++i) {

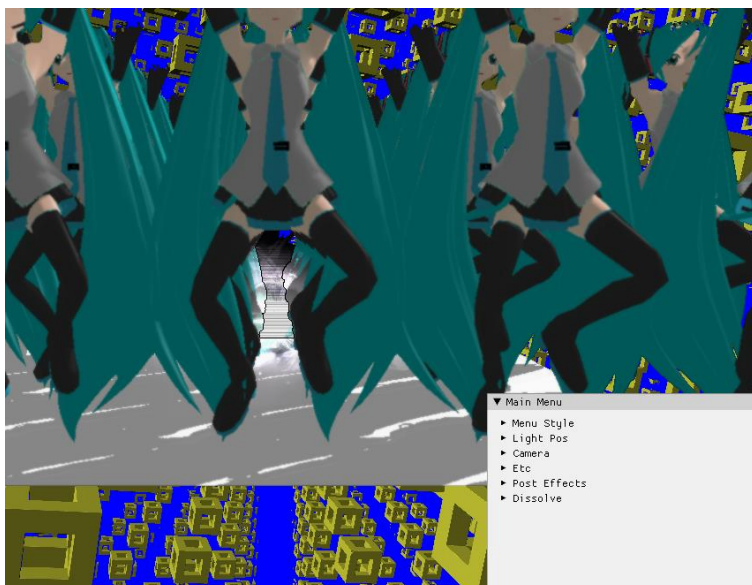
    float sdf = SDFBoundingBoxLattice(start, 0.2, 0.05, 2.0f); // SDFRoundBox2(start, float3(0.2, 0.2,
0.2), 0.1); // SDFSphereLattice(start, 0.25, 1.0, n);
    if (sdf <= epsilon) {
        float3 epsx = float3(epsilon, 0, 0);
```

```

float3 epsy = float3(0, epsilon, 0);
float3 epsz = float3(0, 0, epsilon);
float3 n = float3(
    SDFBoundingBoxLattice(start + epsx, 0.2, 0.05, 2.0) -
SDFBoundingBoxLattice(start - epsx, 0.2, 0.05, 2.0),
    SDFBoundingBoxLattice(start + epsy, 0.2, 0.05, 2.0) -
SDFBoundingBoxLattice(start - epsy, 0.2, 0.05, 2.0),
    SDFBoundingBoxLattice(start + epsz, 0.2, 0.05, 2.0) -
SDFBoundingBoxLattice(start - epsz, 0.2, 0.05, 2.0));
n = normalize(n);
float3 mlight = float3(-1, 1, -1);
float dotVal = saturate(dot(n, normalize(mlight)));
float b = saturate(dotVal + 0.1);
float3 halfV = -ray + normalize(float3(-1, 1, -1));
halfV = normalize(halfV);
float3 mcol = float3(0.8f, 0.8f, 0.2f); //元の色
mcol *= b;
mcol += pow(saturate(dot(halfV, n)), 30);
return float4(mcol, 1);
}
start += ray * sdf;
}
return float4(0, 0, 1, 1);

```

とか書けば



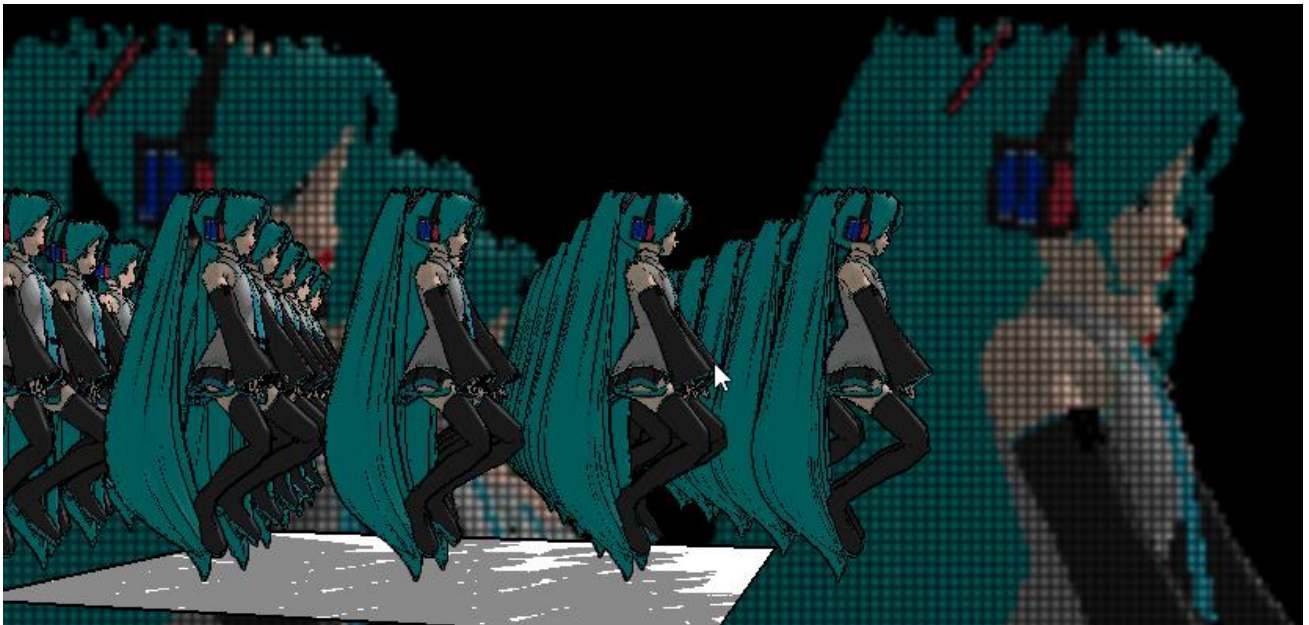
こんな感じの、なんか中に穴の開いたキューブが背景に無数に出てくる。

その他背景にぎやかし活用法

あと、背景の活用法で、レイマーチングじゃないですが、オフスクリーンに書いたものを拡大して、電光掲示板みたいにして表示したりもできます。

```
float4 bcol=rtvTex.Sample(smp, input.uv * 0.25 + float2(0.3, 0.2));  
float2 div = float2(float(w)/float(h), 1.0f)*100.0f;  
float2 ppos = fmod(input.uv, 1.0f / div) * div;  
float rd=1-distance(float2(0.5, 0.5), ppos);  
if (rd > 0.2) {  
    return float4(bcol.rgb * rd*b, 1);  
}  
else {  
    return float4(0, 0, 0, 0);  
}
```

これだけ。ちょっとマジックナンバー吐いてるのでアレですが、これで



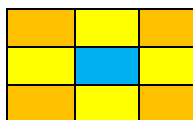
このようなことができるので、工夫次第でいろいろできちゃうと思います。

輪郭線

スプライトアウトライン



こういうのを作りたいとします。で、先ほどのレイマーチングというか距離関数の考え方を応用します。



はい、ご存じだと思いますが、青いセルから見ると、黄色は4近傍です。オレンジのを含めると8近傍なのですが、ピクセルシェータは各ピクセルを塗りつぶす際にそこを何色で塗りつぶすかを決定するものです。で、この例の場合は、8近傍以内に『元の画像のピクセル』言い換えると $\alpha \neq 0$ の部分があれば青で塗りつぶしています。

というわけでこういう関数を作ります。8近傍をチェックして最も大きい α 値のものを探しています。

```
float OutlineCheck(float2 uv){  
    #define DIV_SQRT_2 0.70710678118  
    float2 directions[8] = {float2(1, 0), float2(0, 1), float2(-1, 0), float2(0, -1),  
        float2(DIV_SQRT_2, DIV_SQRT_2), float2(-DIV_SQRT_2, DIV_SQRT_2),  
        float2(-DIV_SQRT_2, -DIV_SQRT_2), float2(DIV_SQRT_2, -DIV_SQRT_2)};  
    float maxAlpha = 0.0f;
```

```

for(uint level=1;level<=1;++level){
    for(uint index = 0; index<8; index++){
        float2 sampleUV = uv + directions[index] * float(level)*0.01;
        maxAlpha = max(maxAlpha, tex.Sample(smp,sampleUV).a/float(level));
    }
}
return maxAlpha;
}

```

探すというより、最も大きい α 値を返してるだけですね。で、これで得た α 値から

```

float maxAlpha=OutlineCheck(PSInput.uv);
PSOutput.Output.rgb = lerp(float3(0.5, 0.5, 1), col.rgb, col.a);
PSOutput.Output.a=max(col.a,maxAlpha);

```

こうします。わかる人にはわかるかと思いますが、2 行目は元の画像の α 値により元画像にするか、青にするかを設定しています(なお lerp は線形補間です)

で、3 行目で、そのまま α 値を使うと輪郭まで α テストで消えてしまうので、先ほどの関数で取得した α 値と比較し、でかいほうを使います(絵がある部分は元絵の α が採用され、元絵の 8 近傍ピクセルでは maxAlpha が採用され、結果的に輪郭線になります)

なお、これの応用編として、雲模様画像と合成するとちょっと面白いことになります。

オーラ

まず、雲模様の画像を

```
Texture2D aura : register( t1 );
```

で参照しておきます。

で、先ほどの距離関数をこの画像でゆがませます。

```
float auraLevel=aura.Sample(smp,uv+float2(0.0f,angle)).r;
```

として置き、先ほどの輪郭線を

```
float2 sampleUV = uv + directions[index] * float(level)*0.01*auraLevel;
```

として少しゆがませます。そうすると



スプライトの廻りになんか邪悪なものが出るようになります。なお、雲模様のUVは動的に変更しておく
と動きが出ていい。

3Dアウトライン

はい、3D空間上でオブジェクトを選択したら、それに輪郭を付けたいことがありますね。今回は『背面法』で
やってみます(2回描画が必要)

と、その前に、モデルに対してシェータを適用しようとするときちょっとややこしいことに、頂点シェータから
いじる必要があります。なので、通常描画とシェータあり描画で分けます。

シェータあり描画の時は

```
DxLib::MV1SetUseOrigShader (true);
```

```
SetUseVertexShader (vs);
```

```
SetUsePixelShader (ps);
```

こんな感じにします。なお、頂点シェータもピクセルシェータも必須です。面倒です。

で、輪郭線を出す手順を言っておくと

①元のモデルを描画

②輪郭線用のモデルを描画

i. カリング面を反転する

- ii. 法線方向に広げる
- iii. 輪郭線色で塗りつぶす

です。カリング面を反転するのはシェータ側じゃないので簡単です。

```
auto num=MV1GetMeshNum(model);
for (int i = 0; i < num; ++i) {
    MV1SetMeshBackCulling(model, i, DX_CULLING_RIGHT);
}
```

簡単といえながら、何このループという事ですが、モデルには複数のメッシュがあり、それぞれに番号が割り振られてますが、全部反転させるためにこうしています。カリングというのは背面を描画しないようにするためのものです。これを反転なんですから、内側だけが描画されるというわけです。

法線方向に頂点を移動させるのはちょっとややこしいです。まず頂点シェータを定義しますが、DxLibの性質上、ここが固定値の上に、ドキュメントがないというぶっ飛びしたくなる状態になっています。

// 頂点シェーダーの入力

```
struct VS_INPUT
{
    float3 Position      : POSITION ;           // 座標(ローカル空間)
    float3 Normal        : NORMAL ;           // 法線(ローカル空間)
    float4 Diffuse        : COLOR0 ;           // ディフューズカラー
    float4 Specular       : COLOR1 ;           // スペキュラカラー
    float4 TexCoords0     : TEXCOORD0 ;        // テクスチャ座標
    float4 TexCoords1     : TEXCOORD1 ;        // サブテクスチャ座標
};
```

// 頂点シェーダーの出力

```
struct VS_OUTPUT
{
    float4 Diffuse        : COLOR0 ;           // ディフューズカラー
    float4 Specular       : COLOR1 ;           // スペキュラカラー
    float2 TexCoords0     : TEXCOORD0 ;        // テクスチャ座標
    float4 Position       : SV_POSITION ;      // 座標(プロジェクション空間)
};
```

// 基本パラメータ

```
struct DX_D3D11_VS_CONST_BUFFER_BASE
{
```

```

        float4x4      AntiViewportMatrix;                                // アンチビューポート行
列
        float4x4      ProjectionMatrix;                                // ビュー → プロジェ
クシヨ ン行列
        float4x3      ViewMatrix;                                    // ワールド
→ ビュー行列
        float4x3      LocalWorldMatrix;                                // ローカル → ワール
ド行列

        float4        ToonOutLineSize ;                                // トウ ー ンの
輪郭線の大きさ
        float         DiffuseSource ;                                // デ
ィフューズカラー( 0.0f:マテリアル 1.0f:頂点 )
        float         SpecularSource ;                                // スペキュラ
カラー( 0.0f:マテリアル 1.0f:頂点 )
        float         MulSpecularColor ;                                // ス
ペキュラカラー値に乗算する値( スペキュラ無効処理で使用 )
        float         Padding ;

};

// その他の行列
struct DX_D3D11_VS_CONST_BUFFER_OTHERMATRIX
{
    float4             ShadowMapLightViewProjectionMatrix[ 3 ][ 4 ] ;                                // シ
ャドウマップ用のライトビュー行列とライト射影行列を乗算したもの
    float4             TextureMatrix[ 3 ][ 2 ] ;
        // テクスチャ座標操作用行列
};

// 基本パラメータ
cbuffer cbD3D11_CONST_BUFFER_VS_BASE                                : register( b1 )
{
    DX_D3D11_VS_CONST_BUFFER_BASE                                g_Base ;
};

```

はい、ここまで固定です。腹立ちますね。で、さらに言うと、カメラ行列とワールド行列が 4x3 になってるのでさらにややこしい。


```

IWorldPosition = float4(mul(ILocalPosition, g_Base.LocalWorldMatrix ), 1.0f);
IViewPosition = float4(mul(IWorldPosition, g_Base.ViewMatrix ), 1.0f);
VSOutput.Position = mul(IViewPosition, g_Base.ProjectionMatrix);

```

こう書いてやらなければいけません。

で、最初のワールドポジションですが、これを法線方向に広げてあげます。

```

ILocalPosition.xyz = VSInput.Position+ VSInput.Normal*8;
ILocalPosition.w = 1.0f ;

```

今回8を書けてますが、適当です。自分で微調整してください。

で、あとはピクセルシェータで指定した色で輪郭線が書かれるわけです。

動的な法線マップの作り方

はい、動的に高さマップから法線マップを作りたい人がいるかもしれません。

レイマーチングの時にもやった偏微分的な考え方で、こうします。

```

const float level = 4.0;
const float eps=1.0f/480.0;
float2 xeps= float2(eps, 0);
float2 yeps = float2(0, eps);

```

```

float3 N=normalize(float3(
    tex.Sample(smp, PSInput.uv + xeps).r - tex.Sample(smp, PSInput.uv - xeps).r,
    tex.Sample(smp, PSInput.uv + yeps).r - tex.Sample(smp, PSInput.uv - yeps).r,
    eps*level));

```

```

N = normalize(N + float3(0, 0, sqrt(1 - N.x * N.x - N.y * N.y)));

```

このNが法線になります。

これを画像にするには

```

PSOutput.Output.rgb = (N + 1) / 2;

```

とするだけです。なお、これは大したことをやってなくて、まず左右のピクセルの濃淡を見て、右が明るければ傾きは正方向です。

上下も同じように考えます。

なお、Z値に謎の値が入ってるのはゼロ除算によってNaNになるのを防ぐためです。

設計とかについて

ひとまずα版終わったあたりで、そろそろ一段落してコードの改善でも行おうかなと思ってるかなと思います。

ます。α版まで作ると全体像が見えてくるので、設計もしやすくなってると思います。

で、なんやかんや言うて、今しばらくは『少なくともゲーム業界においては』オブジェクト指向設計がまだまだ主流かなと思います。いや、もうホントに〇〇界限とかだと『オブジェクト指向はオワコン。まだ C++で消耗してるの?』など割とボロっカスに言われていますが耳を貸してはなりません。

いや、『オブジェクト指向こそ至高!!他はクズ!!!』というわけでもありません。オブジェクト指向は様々な問題を孕んでいるのは事実です。しかし C++を使う業界…特にゲーム業界においてはパフォーマンスを操る必要があるため C++がこのまま生き続けるでしょう。

もしかしたら C#や Rust にとって変わられるかもしれませんが、結局はオブジェクト指向言語です。ちなみに Rust は正確にはオブジェクト指向言語というよりも『マルチパラダイム志向』とも言えます。

マルチパラダイムとは、複数のこういう〇〇志向を組み合わせたものです。

- オブジェクト指向プログラミング
- 関数型プログラミング
- ジェネリックプログラミング

で、おおスゲーと思うのかもしれませんが、これは実は C#にも C++にもある特徴です

- オブジェクト指向プログラミング(ポリモーフィズム)
- 関数型プログラミング(ラムダ式)
- ジェネリックプログラミング(テンプレート)

世の中には『だせえ C++言語を Dis る俺がっこい!!』みたいな思想のおっさんが多く、よく Dis ってきますが、ありゃ単なるカッコつけです。こういう状況を知らずに言ってる恥知らずです。

たしかに C++はメモリ部分を直接いじれるので、危険な言語ではあると思います。しかしそれは裏を返せばメモリを理解してテクニックを使うことで超パフォーマンスを上げたり、割ととんでもないこともできたりするので、少なくともゲーム業界ではそうそう廃れないでしょう。

まあ話がそれましたが、そんな C++の特徴としてはやっぱりオブジェクト指向って事です。前にも話しましたが、この中で一番大事かつ一番ややこしい概念が『ポリモーフィズム』です。

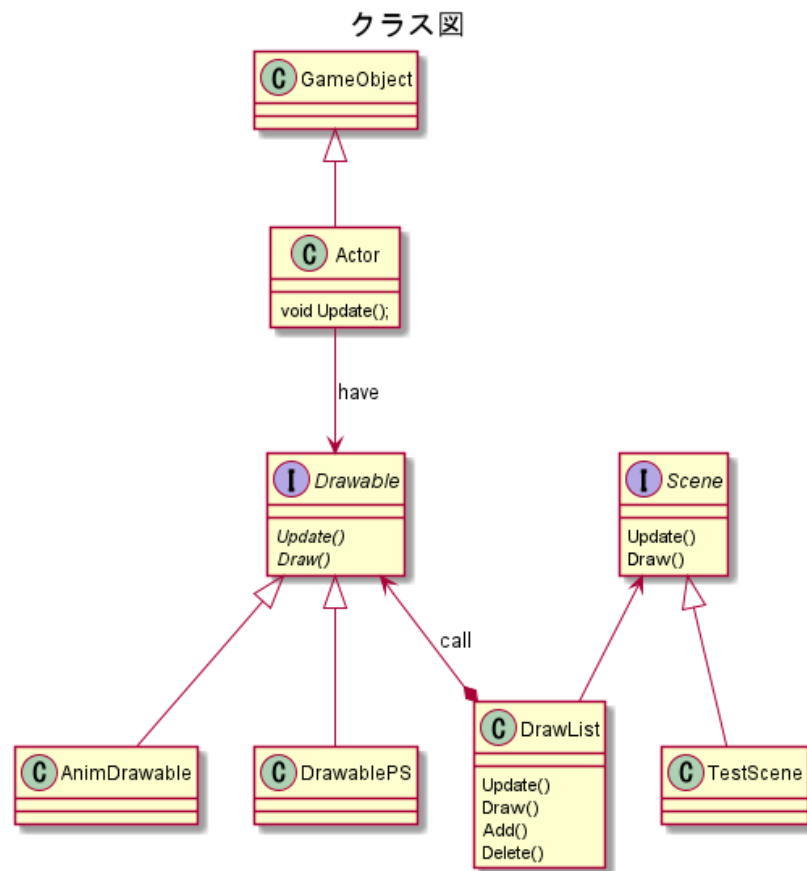
オブジェクト指向の三大要素がありますが、とにかく設計においてはポリモーフィズムの理解の必要があります。

継承による再利用?カプセル化?そうじゃない…とにかくポリモーフィズムだ!!

さてたとえばデザインパターンと呼ばれる者の大半はこの『ポリモーフィズム』を役立てるための設計になっています。

ダイアグラム(クラス図)を書いてみよう

まあ専門的なことはともかく、図を描く習慣をつけてみよう。



で、この図はどうやって書いてんだ？って話ですが、PlantUML というツールを使って書いています。今は便利なもので、VisualStudioCode にインストールして、適切な構文を書いてプレビューすればこれを書き出すことができます。

あ、ファイルに書き出すには Ctrl+Shift+P を押して、テキスト入力欄に "Export Current Diagram" で保存できるぞ。

なお、VisualStudioCode はいいぞ。通常の VisualStudio がカバーしていないようなマイナーなローカルな文法にも対応しているぞ(要個別プラグインインストール)。

例えば上記のようなダイアグラム作成ソフトにも対応しているし xml(gltf)を見るのもできる。もちろん解釈を自分で作ることもできるので、ホンマにマイナー中のマイナーにも対応できる(ただしテキストに限る)

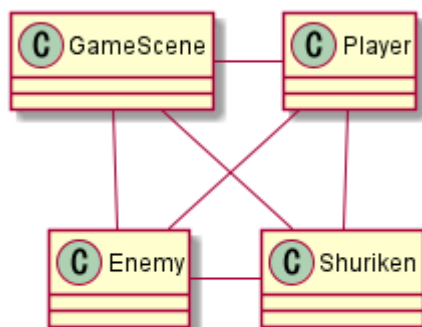
来年度は学校 PC にもインストールするようにお願いしよう。

で、ちょっと話はそれましたが、こういうクラス図を書いておくのは面倒ですが必要です。ちなみに Doxygen を応用すれば自動でこういう図を描いてくれたりもしますが、あまり品質は良くないのと、最初のほうは

自前で書いたほうが練習になると思います。

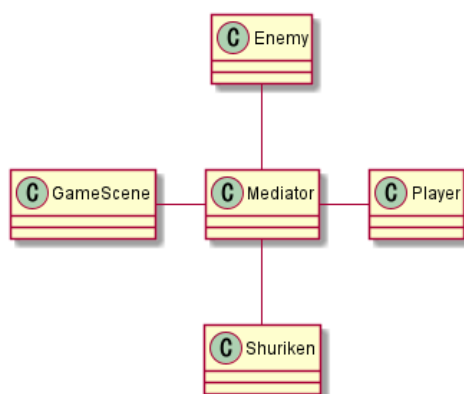
で、なんでこういう図を描いておくといいいのかというと、まず設計の際に妙な設計をしなくなります。あと、クラス間の問題点がわかり、リファクタリングもしやすいです。

例えばクラスが以下のような関係になってたとします



まあやりがちな設計ですね。僕もよくこうなります。しかしこのような関係は一般によくないとされています。簡単に言うとどれか一つ変更すると全体に影響する。全体を変更する必要がある。みたいになります。こういうのを『結合度が高い』と言って、ソフトウェアの品質に影響が出ます。で、まずこういう関係を見つけ出すのが図の役割です。

そしてどうやって行くかということ、ここで登場するのが仲介者パターンってやつで



図のように仲介者(Mediator)を挟むことで、お互いの結合度は緩くなります。このやり方はいわゆる Mediator パターンと呼ばれるものです。前のダイアグラムのように関係が絡まってるのは良くないとされます。

ただ、この場合の Mediator パターン。図で見るほど簡単なわけではなく、ガチで実装しようとすれば色々面倒なことがあるし、必要な事前知識もあるので今は一例くらいに思っておいてください。

サンプルクラス図

ちなみに先ほどまでのダイアグラムのサンプル書いときます。色々文法忘れるんで…最初のやつがこれ↓

@startuml

title クラス図

interface Drawable

```
interface Drawable{
    {abstract} Update()
    {abstract} Draw()
}
```

Drawable <|-- AnimDrawable

```
class AnimDrawable{
}
```

Drawable <|-- DrawablePS

```
class DrawablePS{

}
```

Drawable <--* DrawList : call

```
class DrawList{
    Update()
    Draw()
    Add()
    Delete()
}
```

' main --> DrawList : call

Actor --> Drawable : have

GameObject <|-- Actor

```
class Actor{
    void Update();
}
```

```
interface Scene{
    Update()
    Draw()
}
```

Scene <-- DrawList

Scene <|-- TestScene

@endum1

はい、ちょっと複雑ですが、まあなんとなく分かるんじゃないかと
次にシーンクラスと入力クラスとかはこんな感じ

@startuml

Input--> "1" Keyboard

Input--> "1" Gamepad

Scene..>Input

SceneManager-->Scene

TitleScene..|>Scene

GameplayScene..|>Scene

GameOverScene..|>Scene

ContinueScene..|>Scene

Scene : Update(Input&)

SceneManager : Update(Input&)

Input : Update()

@endum1

次はあやとりみたいになってるクラスですが
何も考えずにこう書きちゃうと

A -- B

C -- D

B -- C

D -- A

A -- C

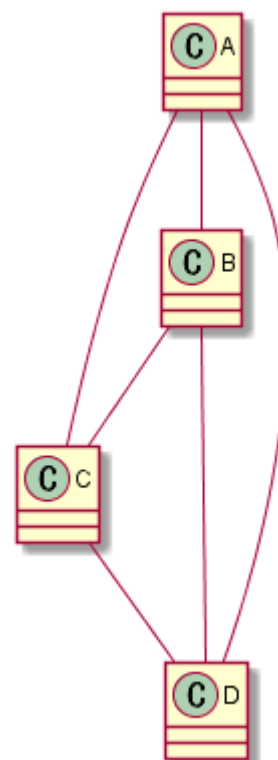
B -- D

右図のようになります。

綺麗に作ろうとすると意外とややこしいんですよ。これが。

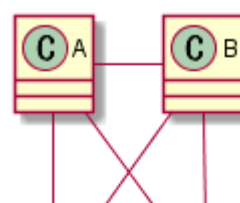
まあ皆さんは関係性がわかればいいので、わざわざ

きれいに書くことはないんですが…



2つポイントがあります。上に書いたものが優先順位が高く、図でも上の方に来ます。そして
-と- ですが、-のほうが強いの結合を表します(ちょっと意味が分かんないっす)
ということで、こう書き直します。

A - B



C - D

A - D

A -- C

B -- C

B -- D

そうすると右図のようになります。

てな感じで工夫してみやすい図を書きましょう。

ちなみに

<|--or<|--

は継承

<--or<--

は方向性を持った依存。相互依存なら<|--を使います。