DX ライブラリでの
エフェクトの作り方

Ver 0.02 4 章までのプレビュー版

前回からの更新点
4 章と数学章の追加。
数学章についてはまだ方針が定まってないので今後結構変わるかもしれないです。

0. 目次

4.パーティクルによるエフェクト
  4-1.パーティクルとは
  4-2.パーティクルその１(円形)
  4-3.パーティクルその２(ヒットエフェクト)
  4-4.炎・煙
  4-5 残像
  4-6.爆発を作る
  4-7.白画像輝度＋パーティクル
  4-8.HSV と RGB
  おまけ 2.外部からフォントを読み込んで使う

M.エフェクトやゲームを作るのに役立つ数学の話
  M-1.直線以外を使うワケ
  M-2.弧度法(ラジアン)と極座標
  M-3.三角関数(sin,cos,~~tan~~ arctan)
  M-4.指数関数
  M-5.イージング関数
  M-6.リサージュ曲線
  M-7.ベジエ曲線

(今後の予定)
5. 自由変形描画とエフェクト
6. レンダリングターゲット
7. UV 座標系

１章：はじめに

プログラム担当がハードコーディングで作るエフェクトというのは前時代的なものになってきたようですが、特に初めて触るときにエフェクトツールとか言われても何がなんだかみたいな感じになったりすることもあるかと思うので、同人ゲームを作るのにおいてこういう情報が一つくらいあってもいいのではないかな、と。

（以下略）

サークル「エンドレスシラフ」及び「CCS」所属
hart

2章：基礎知識など

2-1. DXライブラリDraw系関数それぞれの特徴

DXライブラリには様々な描画関数がありますが、本まとめで使う関数とその特徴を書いていきます。（より詳しい使い方はリファレンスを参照して下さい）

・**DrawLine / DrawBox / DrawCircle**：
左から線、四角形(長方形)、円を描画する関数です。
リファレンス参照。
余談ですが三角形や自由な四角形を描画する関数もあります。

・**DrawGraph**　：画像左上の位置を指定して描画する関数です。
特に他の指定はできません。

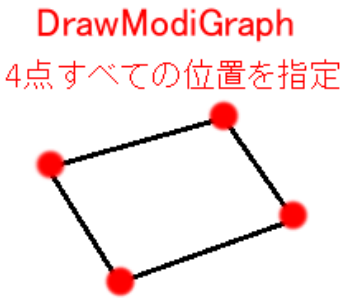
・**DrawRotaGraph**　：画像の中央の位置を指定して描画する関数です。
拡大/回転/左右反転ができます

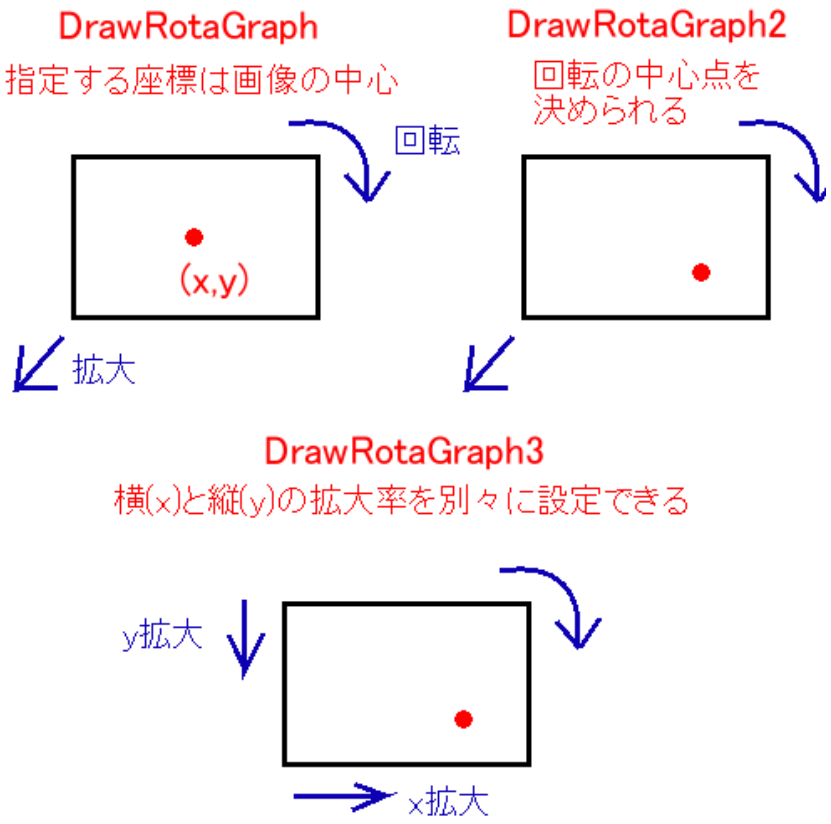
・**DrawRotaGraph2**　：DrawRotaGraphに加えて、回転と拡大の中心を
指定できるようになった関数です。

・**DrawRotaGraph3**　：DrawRotaGraph2に加えてさらに、
X方向、Y方向の拡大率を別々に選べる様になった関数です。
(DXライブラリ3.08c以上で使用可)

・**DrawModiGraph**　：画像を描画する四角形の頂点4つすべての座標を指定しないといけないが、その分自由に変形して描ける関数です。

それぞれの特徴を図に描くと次のようになります。





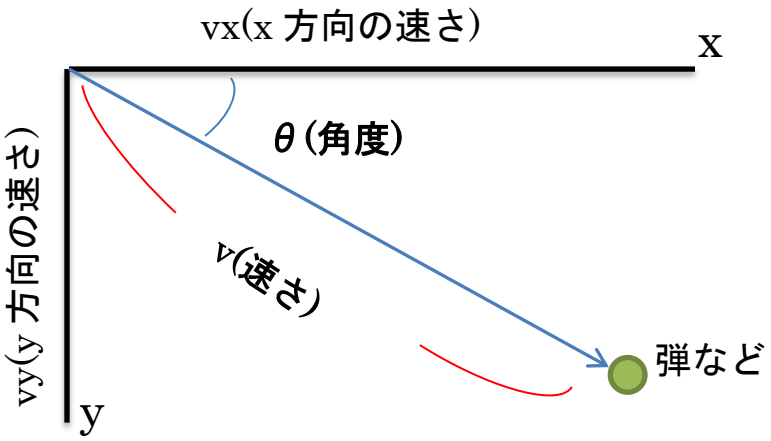
いろいろ上げましたが、とりあえずどれ使えばいいのっていう人は
DrawRotaGraph だけ覚えておけばいいと思います。
それぞれの詳しい使い方は本家リファレンスを見るか、
サンプルコードで実際に使われている所を見て下さい。

このまとめ中で特に何も言わずに「回転する」「拡大する」といった場合は大体
DrawRotaGraph 関数を使えば何とかなります。

2-2. 三角関数と位置・速度(触りだけ

はい、三角関数というと sin(サイン)、cos(コサイン)、tan(タンジェント)ですね。
といっても 2D ゲーム等で軽く使う分にはほぼ意識しなくていいです(多分)。
ちょっとややこしいですがサクッと使い方だけ見てしまいましょう。

STG などの弾がある方向に飛んで行くとすると左上原点の DxLib では



このようになります。
STG における弾等、物体の動きは最低限
「x 方向の速度と y 方向の速度」
または
「速さと角度」
があれば計算出来ます。

例えば弾の座標を(x , y)、弾が 1 フレームに動く距離を(vx , vy) としますと、
弾を動かすには

```
x += vx ;　　y += vy ;
```

とすればいいですよね。
この時の vx,vy を自分で頑張って設定すればいいわけです。
が、これでは狙った方向に動かすのは難しいはずです。
大抵の場合で vx,vy は自分で設定するのは難しいような小数の値になっているからです。

ここで、三角関数が登場します。
この場合は vx , vy を以下のようにして計算します。
x 方向の速度(vx) ＝ 速さ(v)×cos (角度(θ))
y 方向の速度(vy) ＝ 速さ(v)×sin (角度(θ))
たったこれだけです。(cos とか sin とかの計算がやっている内容はわからなければ今はどうでもいいので、結果だけを利用させてもらいましょう)
つまりコードはこのような感じになります。

```
vx = velocity * cos( angle )
vy = velocity * sin( angle )
x += vx ;
y += vy ;
```

もう一つ、例えば STG で自機狙い弾を作るとしましょう。
この時、自機狙い弾の位置(x,y)と自機の位置(px,py)が取得できるとすると、
自機狙い弾が自機の方に向かう角度はアークタンジェント(atan)を使って
簡単に計算出来ます。
(自機の方向の角度) = atan2 (py – y , px – x)
ひとまず上の太字３つの式だけ覚えておけばなんとかなります。

※注意点
三角関数は Math.h をインクルードしないと使えません。
#include <math.h> をプログラムの一番上に書いておきましょう。

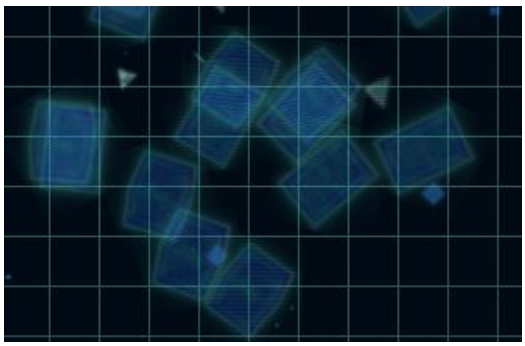
※注意点２
cos/sin 関数の引数及び atan2 関数の戻り値は「弧度法(ラジアン)」での角度になります。
そのため、普段使っている、例えば 45 度を指定したい場合、
cos(45 * 3.141592(円周率) / 180)
としなければなりません。
同様に、atan2 の返り値が何度か知りたい場合も
atan2(～,～) * 180 / 3.141592(円周率)
とする必要があります。

弧度法と三角関数は数学章である M 章で再度軽く触れますので気になる方はそちらをご覧下さい。

2-3. 透過/加算/減算/反転合成

ブレンドモードはエフェクト作成において非常に重要となります。
DX ライブラリではブレンドモードの切り替えは SetDrawBlendMode で行いますが、詳しくはリファレンス参照のこと。
ここではその概要のみ説明していきます。

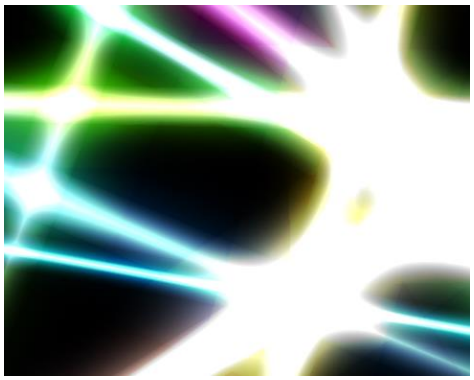
○透過合成(αブレンド)（SetDrawBlendMode 引数→DX_BLENDMODE_ALPHA）
その名の通り画像を透過させることができます。



←このように後ろが透けて見えるように描くことができます。

不透明から透明に徐々に値を変えていくことで徐々に消えていくように見せたりなどエフェクトにとっては非常に重要です。

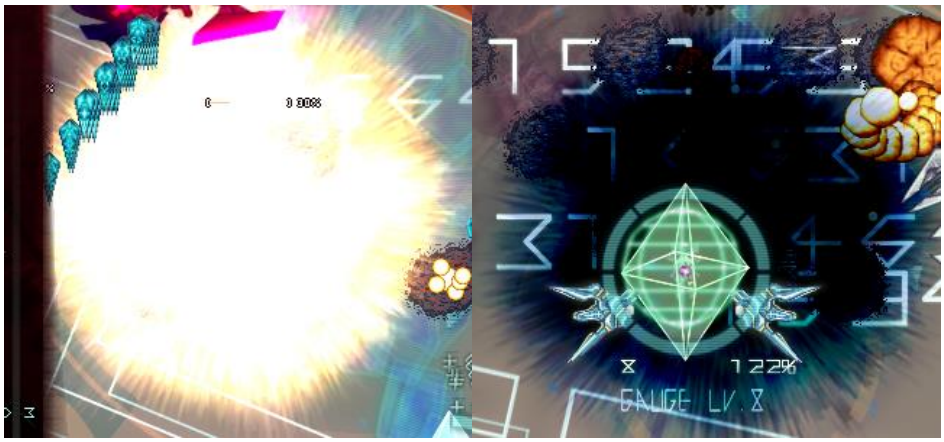
○加算合成(加算ブレンド)(SetDrawBlendMode 引数→DX_BLENDMODE_ADD)
次に描く色を元からある色に加算して描画するモードです。
色は三原色である RGB(赤、緑、青)の組み合わせによって表現できるので、それぞれ 255 が最大値とすると、
例えば赤色(R:255 , G:0 , B:0)が書いてある部分に
緑色(R:0 , G:255 , B:0)を加算合成した場合、
黄色(R:255+0 , G:0+255 , B:0+0)
が結果的に画面に表示されることになります。
加算するという性質上、白(R:255,G:255,B:255)に近づいていくためたくさん重ねると真っ白になっていく(光っているように見える)という効果があります。



←加算合成。
光っているように見えるという特性上、
エフェクトとしては使いやすくかっこいいが
過度な多用は禁物。(詳しくはおまけ 2 を参照してください)

○減算合成(減算ブレンド)(SetDrawBlendMode 引数→DX_BLENDMODE_SUB)

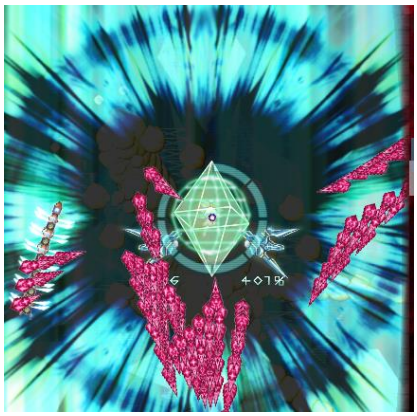
加算合成の逆です。
こっちは新しく重ねた色分だけ元からあった色の値を引きます。
要するに重ねると黒(R:0 , G:0 , B:0)に近づいていく処理ですがあまり使用機会は多くないかもしれないです。



↑加算合成の爆発(左)と減算合成の爆発(右)
違いがよくわかると思います。(真っ白になる加算合成と真っ黒になる減算合成)

○反転合成(反転ブレンド) (SetDrawBlendMode 引数→DX_BLENDMODE_INVSRC)

その名の通り RGB 値を反転させて描画します。



←爆発が反転合成。
減算合成の時に示した画像と同じ画像を反転合成で描いています。

こちらも使用タイミングは微妙なところかもしれないですが紹介程度に。

ここまで 4 つ紹介して来ましたが、
透過と加算は最低限覚えておいたほうが良いです。

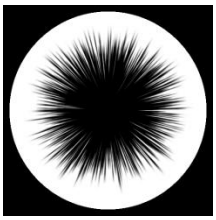
2-4. 輝度

DX ライブラリには表示する画像の明るさを RGB それぞれの色について 0～255(100%)で指定する関数があります。→ **SetDrawBright**
これを使用すると、画像を暗くして表示したり特定の色成分だけで描画したりできます。

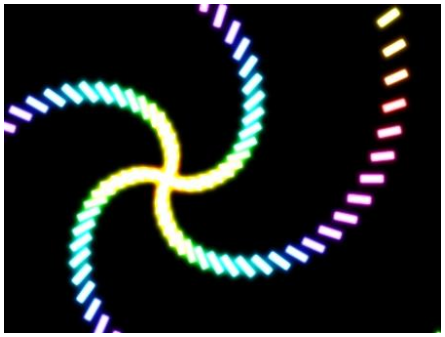
例えば、「無敵時にキャラが赤く点滅する」といった処理は定期的に描画輝度全色 100%と赤(R:255 , G:0 , B:0)を切り替えることで実装できます。

↑左から輝度全色 100%、R のみ 100%(赤)、GB が 100%(水色)、全色 25%(暗く)。
こんな感じになります。

と、普通ならこれだけなのですが、例えば右の画像のように
元画像を白色(R:255 , G:255 , B:255)だけで作ったような
画像を用意しておくと、輝度で指定した色そのものが画像の色に
なるので好きな色で描画するといったことができます
(4 章 7 節を参照)。



実際に前節で紹介した加算合成などと輝度を使うと下の画像のような感じのエフェクトが真っ白な画像だけから作れたりします。



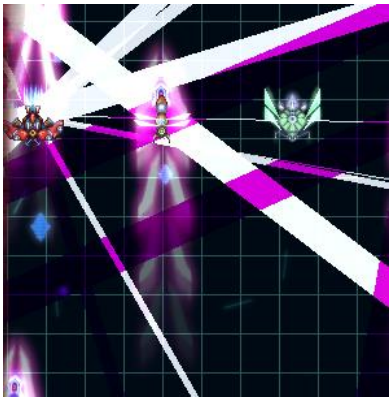
前節ほど重要ではないものの、描画輝度変更もうまく使うと非常に強力です。
(余談ですが一番右の軌跡エフェクトは DrawLine 関数と輝度変更関数だけで作っています。ちょっと重いので多用は禁物ですが)

３章：ごく簡単な２Ｄエフェクト

ここから実際にエフェクトを作っていきます。

3-1.DrawLine での斬撃

まず最初に、素材がなくても作れるエフェクトとして元々ある描画関数である DrawLine を使ったエフェクトを見ていきます。



簡単な斬撃エフェクトということでやり方としては
①画面全体を貫くように太い線を描く
②時間経過ですぐに細くなって消える
ということをやると右の画像のようになります。

実際にやるには DrawLine を使うだけですが、太くするというのをどうやるかが問題になります。
そこでリファレンスには書いていないのですが、DrawLine の６つ目の引数を使うとこの問題を簡単に解決することができます。
DrawLine の引数は以下のとおり

DrawLine(int x1 , int y1, int x2 , int y2 , int Color , int Thickness = 1)
最後に見たこと無い引数があると思いますが、ここの値が線の太さを指定する部分になっています。(いつも書かなくていいのは省略すると自動的に 1 が指定されるようになっているからです。)

最終的にどうすればいいかというと、
「DrawLine で Thickness の引数を最初は大きく、時間経過で徐々に小さくする」
ということをすると簡単な斬撃エフェクトを作ることができます。

このようにアイデア次第では画像がなくても普通に使えるようなエフェクトを作ることができます。

3-2. 画像側でのアニメーション

画像に全く頼らないエフェクトを見たところで、画像側でのアニメーションも見ていこうと思います。
画像でないと作れないエフェクトも結構あったりするので簡単ですがかなり重要です。
画像を作る技術はまた別のお話なのでそこは省略させて頂きますが。

用意した画像はこちら↓(爆発エフェクトですね)

やり方としては
①配列を作る
②LoadDivGraph で分割して読み込む
③Draw 関数で時間に合わせて表示する画像を変えていく
となります。
上の画像の場合の中核となるコードは

```
//読み込み
int explode[12];
LoadDivGraph( "exp.png" , 12 , 6 , 2 , 32 , 32 , explode );
//中略
//表示( timer が時間経過で増える変数 )
int frame = timer % 12;     //余りを使って配列の添字を超えないように
DrawRotaGraph( x , y , 1.0 , 0 , explode[ frame ] , TRUE );
```

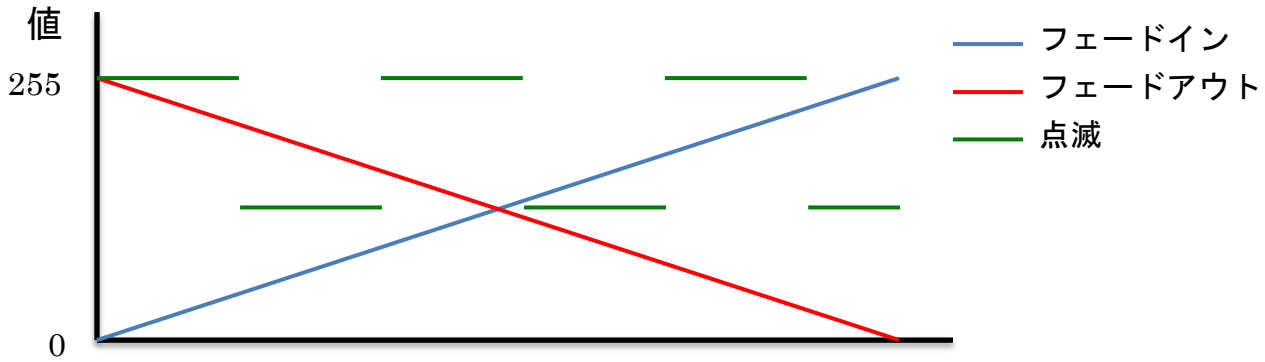
こんな感じになるかなと。
詳しくはサンプルプログラムを見て下さい。

LoadDivGraph が使えるのは上で使った画像のように同じサイズの画像が複数並んでいる時だけなので、右のようにサイズがバラバラの場合は、LoadGraph と DerivationGraph を使って読み込むことになります。
そもそもの画像側で読み込みやすいように調整しておくと便利です。

3-3. フェードイン/アウト、点滅

徐々に現れたり、消えたり、点滅したりといった効果の作り方です。

手順は、「SetDrawBlendMode(DX_BLENDMODE_ALPHA , 値)の値の部分を時間によって変化させる」だけです。
値は 0 で完全に透明(不可視)、255 で完全に不透明なので
入れる値の変化としてはおおよそ次のようになります。

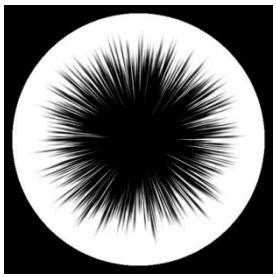


・フェードイン
　時間に合わせて最大値まで増加させていく
・フェードアウト
　時間に合わせて最大値から減らしていく
・点滅
　一定時間の間隔で大きい値と小さい値を切り替える。
　点滅は画像を描画する/しないでも作ることができたり、輝度を設定して
　一定時間ごとに赤く点滅するように見せたりすることもできます。

輝度を使って点滅させるとこんな感じに。

3-4. 簡易衝撃波

さっそく前節のフェードを使って衝撃波を作ってみます。
画像はこれ↓



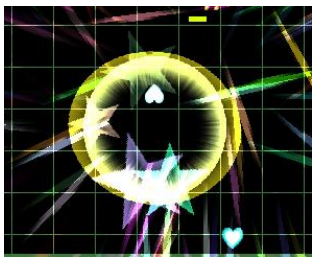
方針としては
①最初は小さく描画し、徐々に大きくしていく
②終わりの方でフェードアウトをかけて消す
の２つです。

コードの核となる部分だけ書きますと、

```
SetDrawBlendMode( DX_BLENDMODE_ALPHA , value );
DrawRotaGraph( x , y , size , 0 , graphichandle , TRUE );
SetDrawBlendMode( DX_BLENDMODE_NOBLEND , 255 );
```

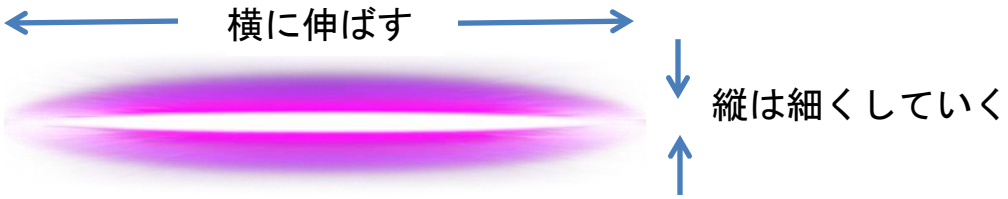
となりますが、重要なのは赤文字にしたアルファブレンドの値と画像の描画サイズです。
アルファブレンドの値は時間とともに 255 から 0 にし、
size の値は適当に拡大していくととりあえずの衝撃波っぽいものはできます。

実際に使うには毎回同じだけの値の変化ではメリハリがない衝撃波になってしまうために、エフェクトの始めの方は拡大スピードを速くして終わりの方ではあまり拡大しないというようなことをしたりするのが必要となってきます。



3-5. 斬撃エフェクトその２

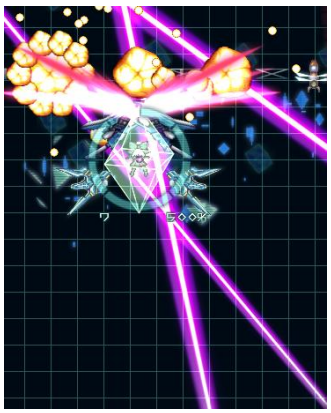
今度は画像を使った斬撃エフェクトを作っていきます。
最初にやった DrawLine と同じで、基本的には「細くしていく」のですが、
今回は「画像を伸ばしながら細くしていく」ことで実装します。



DX ライブラリ 3.08c で追加された DrawRotaGraph3 を使うと簡単に作れます。

```
DrawRotaGraph3( x , y , cx , cy , mag_x , mag_y , angle , grhandle , TRUE );
```

mag_x , mag_y がそれぞれ横(x 方向)への拡大率、縦(y 方向)への拡大率となっています。
mag_x を時間に合わせて大きく、mag_y を 0 に近づけるとそれっぽくなります。
(※画像の中心を指定する cx,cy は元の画像の横と縦のサイズの半分を入れておきましょう)



なんかこんな感じに。

3-6. 文字の強調

文字の強調と言っていますが、要するに下の画像のように同じ画像が拡大＆透過されて広がってくような感じのあれです。(言い方がわからないので実際にサンプルを見たほうがいいかもしれません)

というわけでこのエフェクトは、
①まず画像を描く
②同じ画像を徐々に透過＆大きくしつつ重ねる
ことでできます。

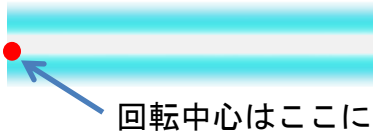
3-7. 直線レーザー

レーザーですが描き方にはいくつか方法があります。

こんな画像が元画像の場合は

A. 同じ画像を複数個つなげる
　→画像がうまくつながるように計算してつなげて表示します
　　(斜めの場合に少し工夫が必要です)

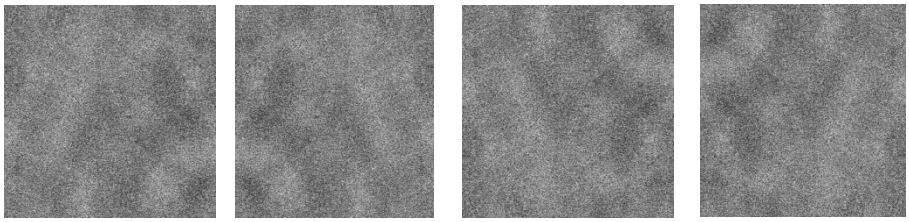
B. 横方向に大きく拡大する
　→DrawRotaGraph3 で中心点の指定を画像の左端に置いて、
　　X の拡大率のみ大きくすることです(右図)
　　関数の値の指定は下のような感じに。




```
DrawRotaGraph3( x , y , 0 ,  画像の高さの半分,
                        , x 拡大率  , 1.0 ,  角度  ,handle
                        , TRUE );
```

3-8. 砂嵐

最近はあまり見ないですが、いわゆるテレビの砂嵐画面です。
要するにランダムな砂嵐画像を複数用意して、
それらを速めの速度で切り替えていけばそれっぽく見えます。
ここでは複数画像用意する代わりに一枚の画像から複数のパターンを作ってみることにします。

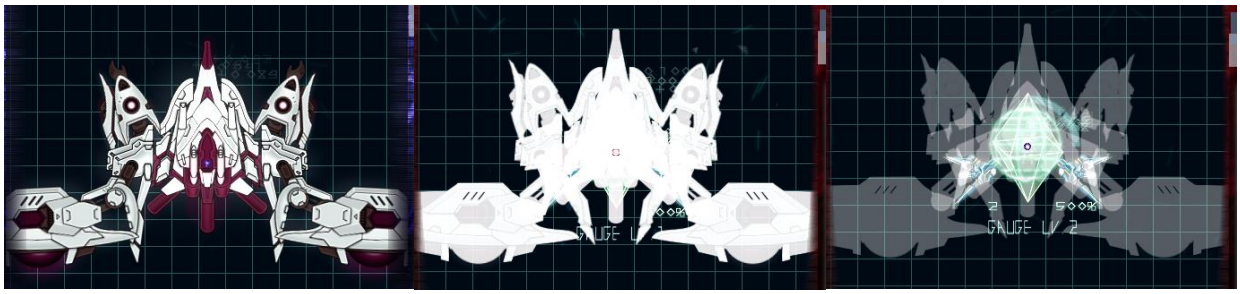


上の画像はそれぞれ一番左の砂嵐を元画像として左右反転、上下反転、上下左右反転をして作ったものです。
これだけでも 4 パターンあるのでランダムに表示すれば砂嵐になります。
簡単に作るには上のように反転したりするのに加えて画像をいくつか用意しておくとそれっぽく見えます。

本来なら完全にランダムでおいたりしたほうがいいのですが、妥協案としては悪く無いかと。

3-9. 画像を真っ白/真っ黒にして消す

画像が白一色になって消えるエフェクトです。



こんな感じに。

やり方は、
①反転合成 100%で元画像を描く
②加算合成を上に重ねる(この時点で全部白くなる)
③加算と反転を同じだけ徐々に値を減らしていく
となります。
コードにすると

```
SetDrawBlendMode( DX_BLENDMODE_INVSRC , 255 ); //反転合成
//Draw関数
SetDrawBlendMode( DX_BLENDMODE_ADD , 255 ); //加算合成
//Draw関数で上と同じ物を重ねる
SetDrawBlendMode( DX_BLENDMODE_NOBLEND, 255 );
```

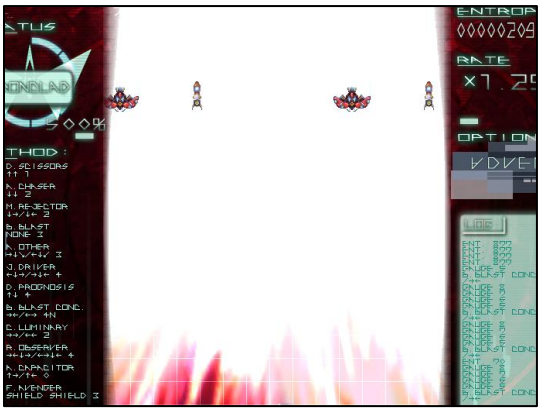
のように書きます。
反対に黒くしたい場合は輝度を RGB 値すべて 0%にすることで黒く出来ます。

なぜ白くなるかということなのですが、
元の色を(r,g,b)としますとまずこれを反転合成で描画した際には(255-r,255-g,255-b)の色で描画されます。
そこに元の(r,g,b)を加算合成するわけですから最終的には(255-r+r,255-g+g,255-b+b)で(255,255,255)の白色が描画されるわけです。

おまけ1. 加算合成の副作用

加算合成はその性質上、合成された部分が白色に近くなっていきます。
それによって明るく見せることができエフェクトがかっこよくなるなどの効果がありますが、逆に言うと「まぶしい」「画面が真っ白になる"白飛び"が発生する」という欠点にもなります。

特に加算合成を常用すると"白飛び"が発生しやすくなるため、多用は禁物です。

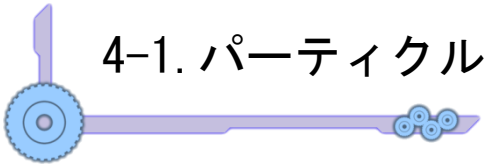


←
加算合成でここまで真っ白になることも。
(これは演出上わざとですが)
ちなみにこの場合は無敵状態になっているので
演出が終わるまで死ぬことはないです。

このように、ともすれば画面上のゲーム上見る必要のある情報すら真っ白にして覆い隠してしまうこともあります。
特に加算合成によって
「STG で弾が見づらい」
「まぶしすぎて自機(または当たり判定)が見えない」
などは「ゲーム性そのもの」に影響を及ぼしてしまっています。
わざとやっている場合はいいのですが、「気がついたら見づらくなっていた」などの状況はできるだけ避けるべきです。

きちんと使えば非常に有用な加算合成ですが、
・そもそもあまり重なりすぎないように注意する
・加算をあまり使わない用に配慮する
・加算によって見づらくなるときは無敵を発生させるなどで
　ゲーム性に影響を及ぼさないようにする
等の配慮が必要かもしれません。

4.パーティクルによるエフェクト



4-1. パーティクルとは

パーティクル(particle)とは英語そのものは「粒子」を意味する言葉ですが、ゲームなどのエフェクトにおいては、「**小さいものを多量に生成することで、炎や爆発などといったものをシミュレートする(またはそれによる独特なエフェクトを創りだす)こと**」と取れます。

パーティクルの考え方を使うことで出来るものとして
・爆発とその破片
・炎や煙
・残像
・光の軌跡
などのゲームには欠かせない多数のエフェクトを作ることが出来ます。
(むしろ大体のエフェクトはパーティクル制御でできていると言ってもいいかもしれません)

そのため、一番重要となるのは
「**どうやって多数のパーティクル一つ一つを制御するか**」
という事になります。

これに関してはシューティングゲームの弾を作ってみるのが一番いいと思われます。
すごく大雑把に言ってしまえば、「パーティクルの基礎は当たり判定の無い弾を複数出す」と見ることが出来ます。
そのため、まずは敵にも自機にもあたらない弾を作ってみればそのままパーティクルに応用出来るのではないかと思います。

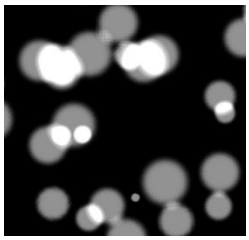
サンプルコードでは簡単に配列と構造体を使って作る予定ですが、リストやC++のクラスといった機能を使うともっとわかりやすくかけるかな、と。(このへんの弾や敵などの管理方法についてはいろいろと資料があると思うのでわからない人は調べてみるといいと思われます。)

4-2. パーティクルその１（円形）

パーティクルを使ったエフェクトの最初として、単に丸い画像を用意してそれを出してみましょう。

やり方は単純に
①パーティクルを多数作る
　作る際に速度と方向をバラバラにして STG の弾と同じ要領で飛ばす
②時間経過でアルファ値(透過度)を低くしていき、一定時間後に消す
となります。

パーティクルにおけるポイントは「多量に出す」ことになりますが、あまり出し過ぎるとくどくなったり、ゲーム性自体に影響がでたり(場合によっては動作スピードにも)するのでそこはうまく考えて使いましょう(3 章おまけ 1 参照)



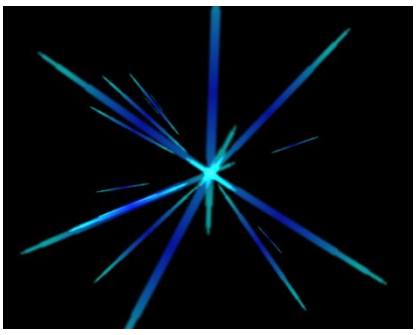
←とりあえず白い画像だけでも動きがあるとそれっぽい感じに。
（静止画だと伝わらない）

4-3. パーティクルその２（ヒットエフェクト）

その１の応用です。
細長い画像を使ってヒットエフェクトのようなものを作ってみます。

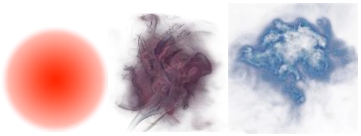

(←画像)

方針は、
① 一度にいろんな方向に飛ぶ細長いパーティクルを
　生成(速度はやめ)
　（生成時の画像の大きさもいじるとそれっぽく）
② だんだん速度を遅くして消す(見栄えのため)
になります。
作った結果は右図に。

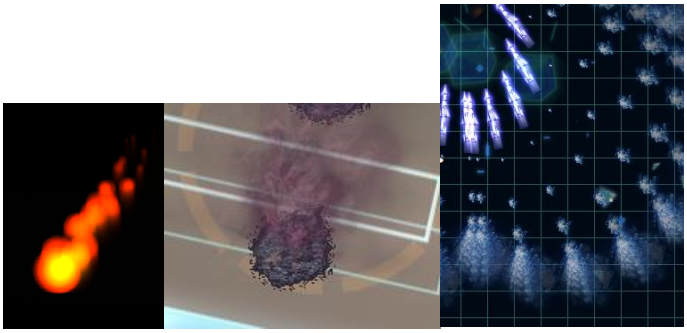


4-4. 炎・煙

ほぼ同じ地点から同じ画像を「ゆっくり上に動かしながら」「拡大しつつ」「透過して消す」と簡単に炎や煙のエフェクトを作ることができます。
今回の画像はこちら



これをパーティクルエフェクトとして一定間隔で発生させつつ、上記のことをすると、

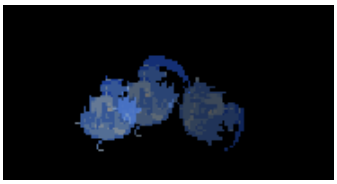


このような見た目のエフェクトを作ることができます。
(一番左だけは加算合成で重ねています)
やることは単純ですがそれなりに見た目はいいです。

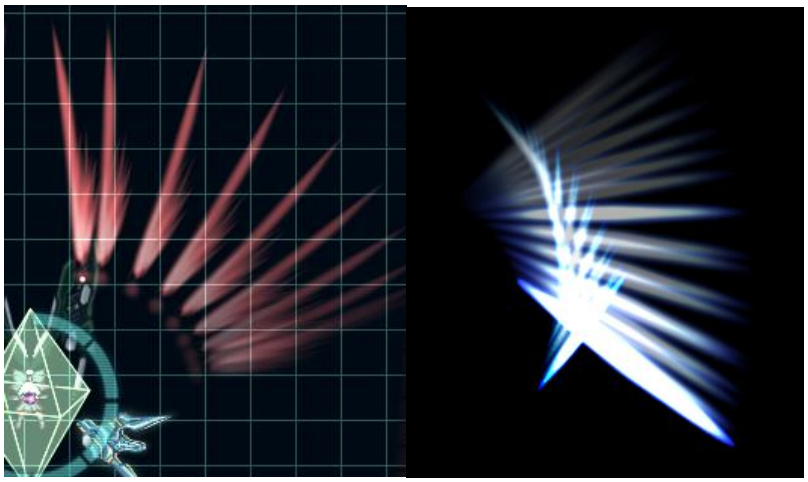
エフェクトを参考にするために見るときは全体として見るのではなく、
・複数のパーティクルに分かれているかどうか
・分かれているならそれぞれがどういう動きをしているか
　┣どの方向にどういう風に移動しているか
　┣どういう合成方法を使っているか
　┗どういう素材からできているか)
が分かるとかなり参考になると思うので、かっこいいエフェクトや参考になりそうなエフェクトを見かけたときはこのへんを気にしてみてみるといいかと思います。

4-5 残像

残像です。
作り方は簡単で、
「残像の生み出し親が一定間隔で自分の今いる位置に一定時間で透過or加算で消える自分と同じ画像を置いていく」
これだけです。
言葉だとわかりにくいので実際どうなるかというと、

キャラに対して使うとこのような感じ。



キャラ以外にでも使えます。
同じ画像をだんだん薄くして表示するだけなので結構簡単に作れます。

4-6. 爆発を作る

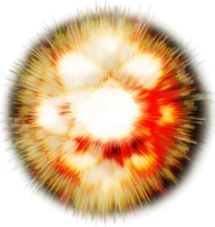
爆発と言ってもいくつもパターンがあるので３つほど。
①画像側でアニメーションを作る

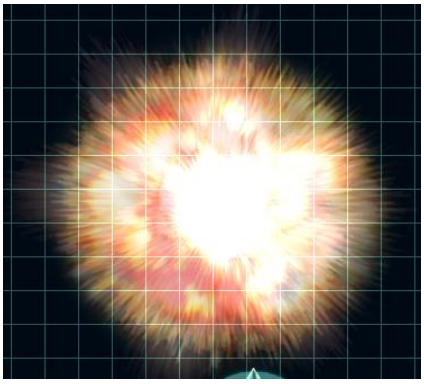
前の章で紹介しましたが、画像さえ作れれば順番に表示していくだけの楽な実装法です。
ゲームの世界観に合わせやすいという利点もありますがとにかく画像の準備がネックとなります。

②衝撃波＋他のパーティクルを飛び散らす

言葉通りです。
結構それっぽくなります。

③他の画像の拡縮でごまかす

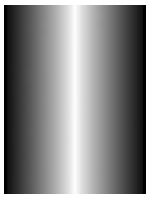
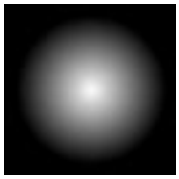




加算合成したり回転したり２つ重ねてみたり・・・
などの工夫をいろいろするとこういうのが出来ます。

4-7. 白画像輝度＋パーティクル

前述の通り、輝度変更を使うと真っ白な画像から色つきの画像として描画することが出来ます。(小技みたいなものですが)
やり方は「表示する元の画像は真っ白で作っておいて、表示する際に輝度を欲しい色に変更してから描画する」となります。


元画像をこのように準備したとします。
実際の描画部分は、例えば赤(R:255 G:0,B:0)なら

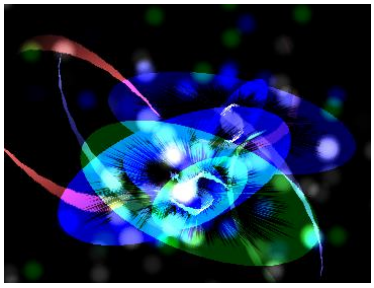
```
//輝度の設定値を赤に変更
SetDrawBright( 255 , 0 , 0 );
DrawRotaGraph( x , y , 1.0 , angle , GraphicHandle , TRUE );
//元に戻す
SetDrawBright( 255 , 255 , 255 );
```

これと加算・透過を使いつつ前節までのパーティクルと組み合わせると、



このような感じのものが作れます
衝撃波と組み合わせると右図のようなことも。

うまく使えば良い感じのエフェクトが作れるかもしれません。



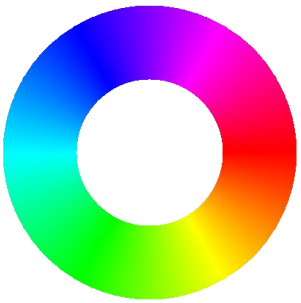
4-8. HSV と RGB

前節で好きな色で描画するというのに軽く触れたのでこの節では虹色を作ってみます。
ここで、赤→橙→黄→緑→青→藍→紫→赤と変化させればいいのですが、普段の RGB での色表現では少々計算が面倒です。

そこで、HSV での色を RGB へと変換して使うことを考えます。

※HSV とは
色相(Hue)、彩度(Saturation)、輝度(Brightness)の 3 成分からなる色表現方法です。
色相は色の種類、彩度は色の鮮やかさ、輝度は色の明るさを示します。
(ペイントソフトの SAI や Gimp を使ったことのある人はカラーサークルでの色指定を思い出してもらえれば)

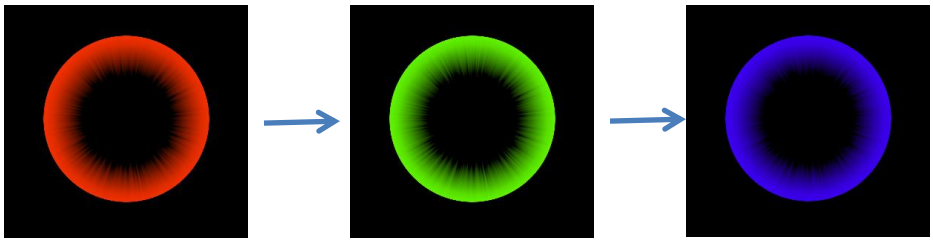
この内今回重要になるのは右図にある
カラーサークルです。
色相(H)の値の変化はこうなっています。



これを見ると分かるように H が 0 から 360 まで
なめらかに色が変わるのと、順番に虹色が
作れます。

今回の方針は「H の値を時間で徐々に変えて(S,V は 1 固定)、得られた HSV を RGB に変換して輝度を変えて描画する」ということになります。
HSV→RGB の変換は「HSV RGB 変換」あたりで検索をかければ出てくるのでそれを C 言語用に書き直せばいけると思います(後ほどサンプルコードにも書きますが)

作り方は
①始め HSV の値を決める(H は時間によって 0～360 の値の範囲で変化)
②HSV の値を RGB に変換する
③変換した RGB を使って輝度を設定する
④元々白い画像をエフェクトとして描画
⑤輝度をもとに戻す。
となります。



なめらかに色が変わっていくので特定の色変化の間の補完とかにも使えないこともないかもしれないです。

おまけ 3. 外部からフォントを読み込んで使う

ちょっとエフェクトから離れた話題を一つ。
DX ライブラリで普通使えるフォントはその Windows にインストールされているものだけなので例えば「自作のフォントを使いたい」といった場合はちょっとした工夫が必要になります。

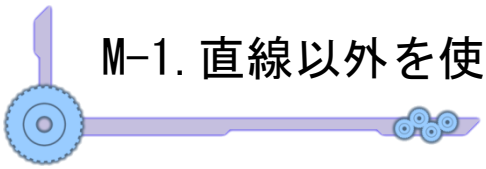
やり方ですが
①読み込みたいフォントデータを ttf ファイルなどで準備する。
②LoadFontResourceEx("ttf ファイルの場所", FR_PRIVATE , NULL)
　でフォントをロード
③DX ライブラリで普通にフォントを使用する
④RemoveFontResourceEx("ttf ファイルの場所", FR_PRIVATE , NULL)
　で読み込んだフォントを破棄
となります。

なお、読み込むフォントファイルは ttf 以外にも読み込めるようです
(MSDN の AddFontResourceEx のページを参照のこと)

↓使うとこうなります。

M.数学の話

数学章です。といっても深い所には突っ込まないので多分大丈夫？



M-1. 直線以外を使うワケ

ゲームを作るだけだったら全ての動きが一定値ずつ増加/減少(グラフにすると直線)で値が変化してもゲームにはなりますし、それで問題ないということもあります。

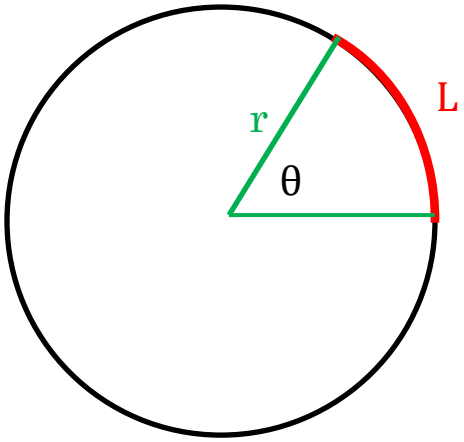
しかし、アクションゲームのジャンプや慣性は加速度運動ですし、シューティングでも最近の弾幕 STG では途中で弾が遅くなったり変な動きをしたりといったものがよく見られます。
それに、エフェクトでも例えば衝撃波が一定速度で拡大するだけではなんか違う印象を受けると思います。(衝撃波は最初大きく拡大して終わり際はあまり拡大しないほうが見栄えがいいです。)

このようにグラフにして直線以外になる色々な挙動(といっても主に加速度運動ですが)を使うことで、それっぽく見せたりするというのは普通に行われています。
(もちろんそれだけが理由ではないですが)

この章では主に「エフェクトや STG などの弾の軌道などで役立つかもしれないような各種曲線」やそもそも「三角関数や弧度法って何か」のようなことを書いていきます。
(あまり深くは突っ込まないようにしようと思っていますが)

M-2. 弧度法(ラジアン)と極座標

普通使っている度数法(45 度など)の他にプログラミングしていると**弧度法(ラジアン)**と呼ばれる角度を使うことがよくあります。

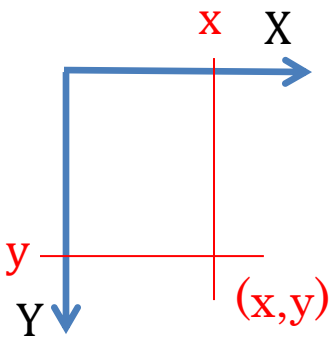


弧度法とは円を考えた時に、
θ(角度) = L(弧の長さ)÷r (円の半径)
で角度を表す方法です
どのくらいの値になるかというと
180(度) == 3.14159265…(ラジアン)
となります。180 度で円周率です。

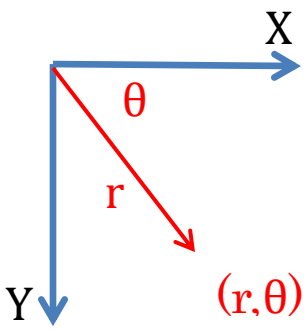
なぜこんな物を使うかというと、
いくつかの関数の引数となっている角度の値が
弧度法の値を取る場合が多いからです。
例を挙げると三角関数(cos , sin)や DrawRotaGraph 系の角度の引数は
ラジアンで渡さなければなりません。
三角関数は何かと便利ですし、DrawRotaGraph もかなり有用な描画関数であるので詳しいことは抜きにしてもラジアンの使い方は知っておいたほうがいいです。

度数からラジアンに変換するには、上で書いた対応を使うと
π(円周率) = 3.14159265…として、
角度(ラジアン) = 角度(度数法) × π ÷ 180
となります。
この計算式を関数またはマクロ化しておいたりすると便利かもしれません。

弧度法に触れたところでもう一つ極座標の話を一応しておこうと思います。
普通扱う座標系は XY 座標系であって、DX ライブラリのウインドウだと X は右に行くほど大きく、Y は下に行くほど大きくなり、X と Y の値を指定するとある 1 点の場所が指定できます。(下左図)
座標を指定する方法として、XY の２つの値を指定する方法の他に、
原点からの距離と角度で指定する方法があります(これを極座標系といいます)(下右図)

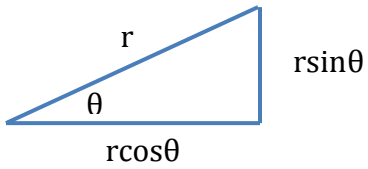


XY 直交座標系



極座標系

ところで、cos と sin は距離 r と角度θに対しそれぞれ下のように対応します。

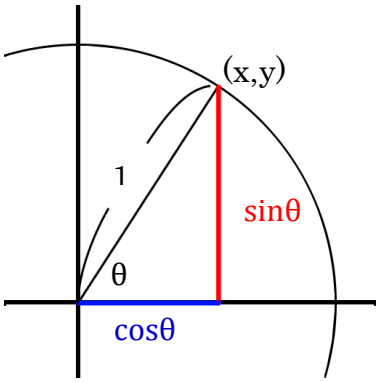


2 章で速度を計算するときに
X 方向の速度 ＝ 速さ × cos (角度)
Y 方向の速度 ＝ 速さ × sin (角度)
と言ったのはこのためです。
ズバリこの式で XY 座標と極座標は対応するので上の図だと
$x = r \times \cos(\theta)$
$y = r \times \sin(\theta)$
というようになります。

M-3. 三角関数

cos や sin などの三角関数はここまででもちょこちょこ出てきていますが、
実際はどういうものかというのに軽く触れておきたいと思います。

いくつか表現方法がありますがここでは半径１の円の時に cos(コサイン),
sin(サイン), tan(タンジェント)のそれぞれの関数がどこを表すか軽く書いておきます。



左図の半径１の円で、円上の点(x, y)を考えると

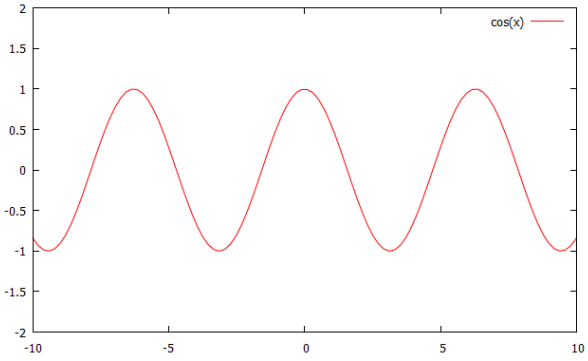
$\cos\theta = x$

$\sin\theta = y$

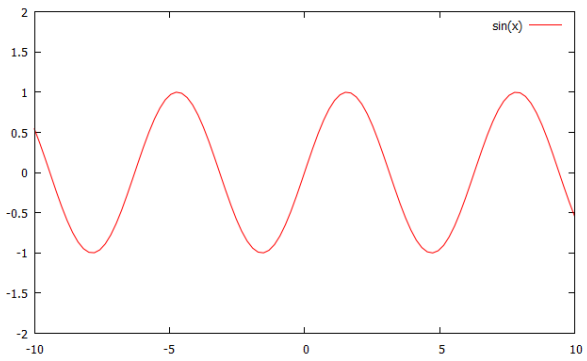
$\tan\theta = \frac{y}{x}$ （ 原点と(x, y)を通る直線の傾き）

となります。

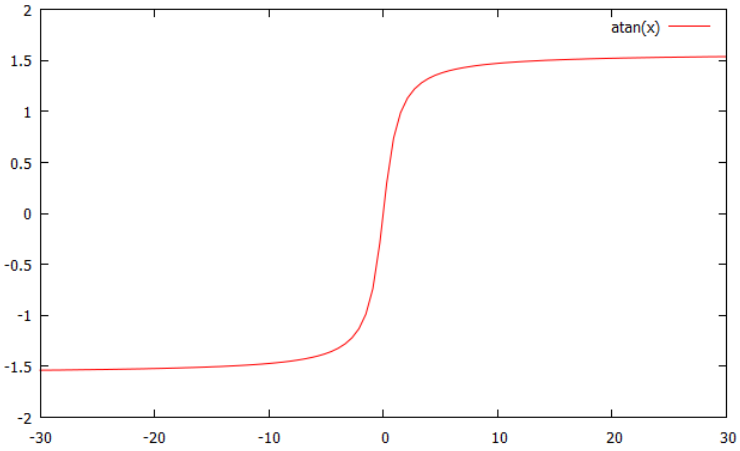
ゲームにおいては３つ目の tan(タンジェント)は atan(アークタンジェント)の形の方が便利なのでそちらも軽く触れておきます。
$y = \tan x$としたときにこれを$x =$ の形に直すと出てくるのがアークタンジェントです。
$x = \operatorname{atan} y$
といってもこれだけではよくわからないので三角関数それぞれについてどういう形状かのグラフを乗せておきます。



↑cos 関数



↑sin 関数



↑atan 関数

cos と sin は x 軸でずれているだけなので形状は同じです。
前に紹介した X 方向の速度、Y 方向の速度を求めるだけでなく、その関数の独特な曲線形状そのものでも使ったりできます。

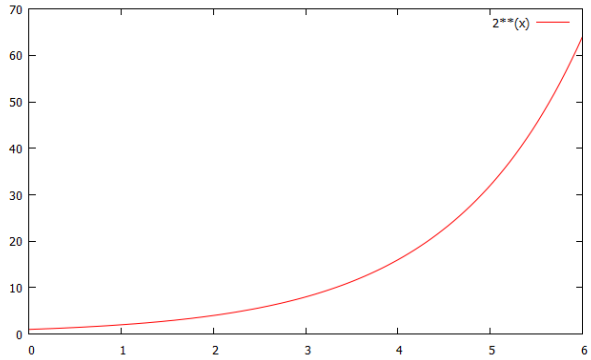
一方 atan ですが、これは STG で自機狙い弾を作ったりするのに活用できます。
自機の位置を(x_player , y_player) として自機を狙っていく弾の現在位置を(x,y)としますと、C 言語では
(自機への方向) = atan2(y_player - y , x_player - x)
で算出できます。(atan2 の戻り値は弧度法の単位で返ってくることに注意)

ややこしいですが実際の中身がどうでもいい人は使い方だけ覚えてしまえばいいのでそんなに身構えるようなものでもないかと思います。
最後にもう一度ゲームで使えそうなものだけ挙げておきます。

X 方向の速度 = 速さ × cos(角度)
Y 方向の速度 = 速さ × sin(角度)
(点 A から点 B への角度)
= atan2(点 B の Y 座標 - 点 A の Y 座標 , 点 B の X 座標 - 点 A の X 座標)

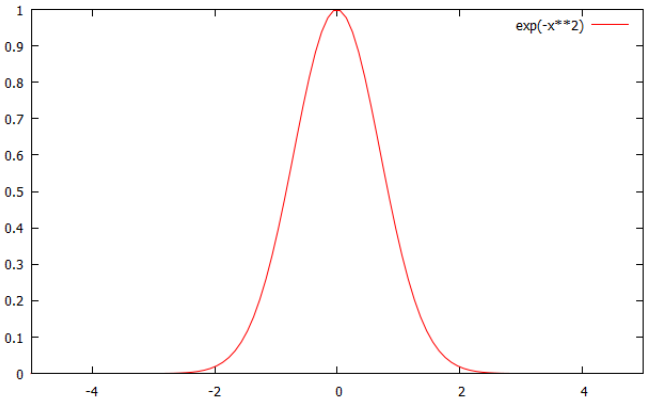
M-4. 指数関数

指数関数は形状としては最初は上がり方が鈍く、x の値が大きくなると
増加が非常に大きくなるという形状をしています。
式は y = n^x と書き、次々に n をかけていく関数ですが、C 言語風に書くと
y = pow(n , x)
となります



グラフ形状としてはこのような形です。

ネイピア数(e)と呼ばれるものと組み合わせて、
y = exp(-pow(x,2))
の形を作ると x でピークを持つ関数になったりします。

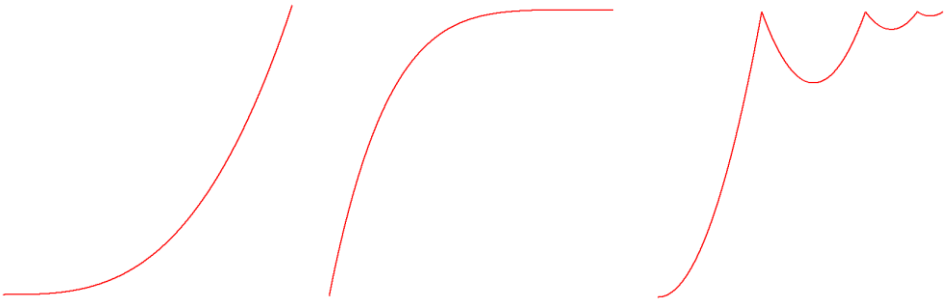


何かに使えるかもしれないので一応参考までに。

M-5. イージング関数について

普通に値を足したりしているだけでは同じ速度でしか動いていかないエフェクトが出来上がります。
エフェクトの見た目は等速で動かすだけじゃなく、衝撃波のように「最初拡大が速くて、後半遅い」動かし方などが重要になってくるものがあります。
自分で加速度などを定義してもいいのですが、イージング(Easing)関数というものがあるので利用してみるのも手です。

JavaScript のライブラリに jQuery というものがありますが、そのプラグインに Easing があり、そこに様々な値の変化をする関数が入っています。
これを参考に C 言語に移植して使ってみるとエフェクトなどの値に緩急をつけるのが簡単にできるのでなかなか便利です。



InCubic　　OutQuint　　OutBounce

試しに InCubic を C 風に書き直すとこのような感じに

```
double InCubic(double t,double totaltime ,double max = 1.0 ,double min = 0.0 )
{
	max -= min;
	t /= totaltime;
	return  max * t*t*t + min;
}
```

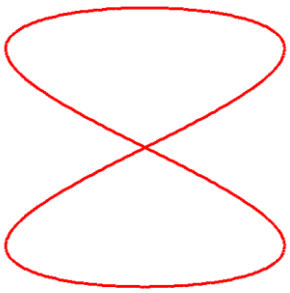
jQuery は原理上オープンソースになっているので興味ある人は見てみるといいかもしれないです。

M-6. リサージュ曲線

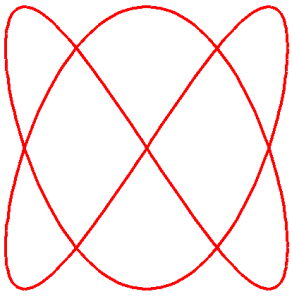
リサージュ曲線は一例として

$$\begin{cases} x = Asin(at) \\ y = Bsin(bt) \end{cases}$$

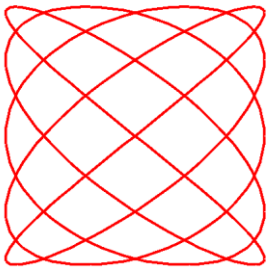
で表される曲線です。A,B,a,b に任意の値を入れることによって様々な形がでます。
よくわからないような軌道が作れたりするのでうまく使うと面白いかもしれないです。
下に一例を乗せておきます(なお、A=B=1 として a,b のみを変化させます)



a = 2 , b = 1



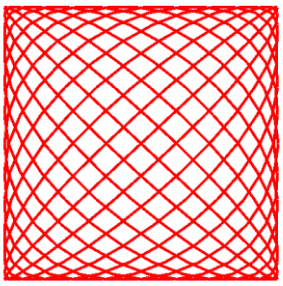
a = 2 , b = 3



a = 5 , b = 4

使い所は限られるかもしれませんが知っておいても損はないのではないかと。
どういう形が出てくるかはやってみないとわからない(予測しづらい)ところが難点ですが。

おまけ↓ : a = 13 , b = 12。もはやよくわかりません。



M-7. ベジエ曲線

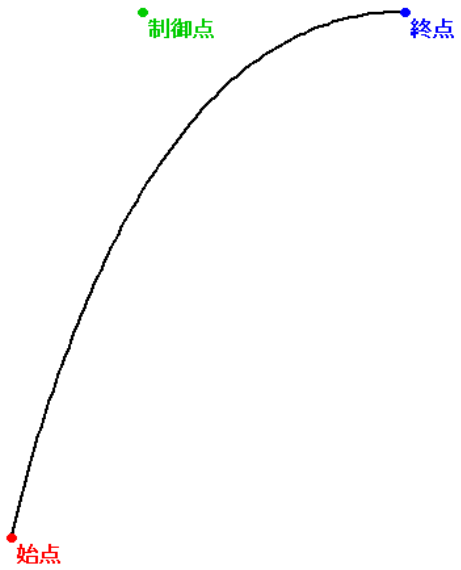
ここでは知っておくと便利なベジエ曲線について軽く触れていきます。
説明の難しい曲線なので詳細は「ベジエ曲線」で検索してください・・・だと投げすぎなので詳しい定義などは省いて概要を書いていきたいと思います。

ベジエ曲線は複数の制御点と呼ばれる点を元に作られる曲線です。
その特性として「始点と終点になる制御点を必ず通る」ことが挙げられます。

ここでは「始点、制御点、終点」の３つの点で作られる 2 次のベジエ曲線を取り上げます。
変数 t を 0 から 1 までの値とすると t=0 で始点を、t=1 で終点を通るベジエ曲線の式は、
始点(sx, sy)，制御点(cx, cy)，終点(ex, ey)として

$$\begin{cases} x = (1-t)*(1-t)*sx + 2*(1-t)*t*cx + t*t*ex \\ y = (1-t)*(1-t)*sy + 2*(1-t)*t*cy + t*t*ey \end{cases}$$

というようにかけます
結果ベジエ曲線は右図のような曲線を描きます。



うまく扱えると制御が楽な曲線であるため意外と便利です。

比較的わかりやすいところでは、Gimp や Photoshop などペイントソフトのパスなどの機能がこの曲線を使っています。
どういう軌道を描くかよくわからない場合はこれらのツールのパス機能などで見てみるといいかもしれません。