[OpenGL 2D 2018 第05回]

広い視野を持って事にあたる

# より広い範囲を調べる

迷宮を探索するのに周囲5マスしか見えないというのでは、いささか心もとないですよね。もう少し遠くまで見えていれば、もっと迷いにくいはずです。そこで、より広い範囲を調べて画面に表示することにしましょう。

## 壁の情報を記録するデータ構造

前回はwall\_left, wall\_right, wall\_frontという3つの配列に壁の情報を記録していました。今回は調べる範囲が広がるため、そのままでは不便です。そこで、壁の情報を記録するデータ構造は2重の配列、つまり2次元配列に変更します。

壁の有無を調べるプログラムの先頭に、次のプログラムを追加してください。

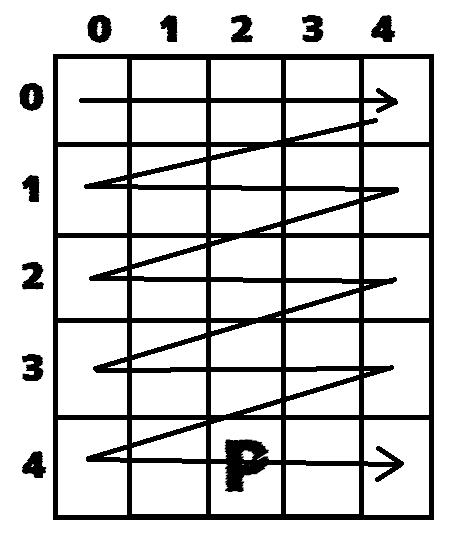
/\*  
 \* ダンジョン探索.  
 \*/  
 reset\_all\_image();  
 reset\_all\_text();  
  
 // 方向に応じて壁の有無を調べる.  
**+** const int check\_width = 5;  
**+** const int check\_height = 5;  
**+** int vicinity\_map[check\_height][check\_width];

check\_width(ちぇっく・うぃす)とcheck\_height(ちぇっく・はいと)が調べる範囲の大きさです。前方4マス、左右2マスずつの範囲を調べることにしました。  
配列変数はvicinity\_map(ゔぃしにてぃ・まっぷ)としました。なおvicinityは「周辺」や「近隣」という意味の英単語です。

次に、壁の情報を調べてvicinity\_map配列に設定するプログラムを書きましょう。  
上記のプログラムの下に、次のプログラムを追加してください。  
やたら長いですが、if～elseで囲まれたよく似たプログラムが4つつながっているだけです。最初の1つだけ書いて残りはコピー&ペーストを使えば、見かけほど大変ではないと思います。

// 方向に応じて壁の有無を調べる.  
 const int check\_width = 5;  
 const int check\_height = 5;  
 int vicinity\_map[check\_height][check\_width];  
**+** if (player\_direction == dir\_up) {  
**+** // 上向きの場合(-2,-4)から右下に向かって調べていく.  
**+** for (int b = 0; b < 5; ++b) {  
**+** const int y = player\_y - 4 + b; // 調査点のY座標.  
**+** for (int a = 0; a < 5; ++a) {  
**+** const int x = player\_x - 2 + a; // 調査点のX座標.  
**+** // 調査点がマップの外側なら壁扱いとする.  
**+** if (x < 0 || x >= dungeon\_width || y < 0 || y >= dungeon\_height) {  
**+** vicinity\_map[b][a] = 1;  
**+** } else {  
**+** vicinity\_map[b][a] = dungeon\_map[y][x];  
**+** }  
**+** }  
**+** }  
**+** } else if (player\_direction == dir\_right) {  
**+** // 右向きの場合(4,-2)から左下に向かって調べていく.  
**+** for (int b = 0; b < 5; ++b) {  
**+** const int x = player\_x + 4 - b; // 調査点のX座標.  
**+** for (int a = 0; a < 5; ++a) {  
**+** const int y = player\_y - 2 + a; // 調査点のY座標.  
**+** // 調査点がマップの外側なら壁扱いとする.  
**+** if (x < 0 || x >= dungeon\_width || y < 0 || y >= dungeon\_height) {  
**+** vicinity\_map[b][a] = 1;  
**+** } else {  
**+** vicinity\_map[b][a] = dungeon\_map[y][x];  
**+** }  
**+** }  
**+** }  
**+** } else if (player\_direction == dir\_down) {  
**+** // 下向きの場合(2,4)から左上に向かって調べていく.  
**+** for (int b = 0; b < 5; ++b) {  
**+** const int y = player\_y + 4 - b; // 調査点のY座標.  
**+** for (int a = 0; a < 5; ++a) {  
**+** const int x = player\_x + 2 - a; // 調査点のX座標.  
**+** // 調査点がマップの外側なら壁扱いとする.  
**+** if (x < 0 || x >= dungeon\_width || y < 0 || y >= dungeon\_height) {  
**+** vicinity\_map[b][a] = 1;  
**+** } else {  
**+** vicinity\_map[b][a] = dungeon\_map[y][x];  
**+** }  
**+** }  
**+** }  
**+** } else {  
**+** // 左向きの場合(-4,)から右上に向かって調べていく.  
**+** for (int b = 0; b < 5; ++b) {  
**+** const int x = player\_x - 4 + b; // 調査点のX座標.  
**+** for (int a = 0; a < 5; ++a) {  
**+** const int y = player\_y + 2 - a; // 調査点のY座標.  
**+** // 調査点がマップの外側なら壁扱いとする.  
**+** if (x < 0 || x >= dungeon\_width || y < 0 || y >= dungeon\_height) {  
**+** vicinity\_map[b][a] = 1;  
**+** } else {  
**+** vicinity\_map[b][a] = dungeon\_map[y][x];  
**+** }  
**+** }  
**+** }  
**+** }

このプログラムは、下図のようにプレイヤー(Pマーク)の左前方からジグザグに壁の情報を取得していきます。横の動きが内側のfor文で、縦の動きが外側のfor文の担当になっています(つまり、横がaで縦がb)。



なお、上記のプログラムを配列を使ってひとつにまとめると、次のようなプログラムになります。

// 方向に応じて壁の有無を調べる(配列バージョン).  
{  
 const int start\_offset[4][2] = { {-2, -4},{ 4, -2 },{ 2, 4 },{ -4, 2 } };  
 const int next\_step\_x[4][2] = { { 1, 0 },{ 0, 1 },{ -1, 0 },{ 0, -1 } };  
 const int next\_step\_y[4][2] = { { 0, 1 },{ -1, 0 },{ 0, -1 },{ 1, 0 } };  
 int x\_base = player\_x + start\_offset[player\_direction][0];  
 int y\_base = player\_y + start\_offset[player\_direction][1];  
 for (int b = 0; b < 5; ++b) {  
 int x = x\_base; // 調査点のX座標.  
 int y = y\_base; // 調査点のY座標.  
 for (int a = 0; a < 5; ++a) {  
 // 調査点がマップの外側なら壁扱いとする.  
 if (x < 0 || x >= dungeon\_width || y < 0 || y >= dungeon\_height) {  
 vicinity\_map[b][a] = 1;  
 } else {  
 vicinity\_map[b][a] = dungeon\_map[y][x];  
 }  
 // 調査点を(プレイヤーから見て)右に1マス移動.  
 x += next\_step\_x[player\_direction][0];  
 y += next\_step\_x[player\_direction][1];  
 }  
 // 調査点を(プレイヤーから見て)下に1マス移動.  
 x\_base += next\_step\_y[player\_direction][0];  
 y\_base += next\_step\_y[player\_direction][1];  
 }  
 }

さきほどと比べるとずいぶん短いですね。これが配列の威力です。興味のある方はif～elseバージョンのどの部分が配列になっているのか調べてみるといいでしょう。また、壁を調べるプログラムにif～elseバージョンではなく配列バージョンを使っても構いません。

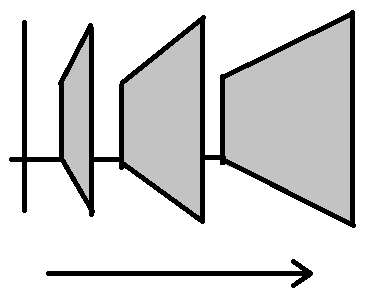
## 壁を描く

広範囲の壁の有無の情報が得られたら、その情報をもとに壁を描いていきます。  
前回は5つのマスごとに専用の画像を用意していましたが、今回のように広い範囲を描く場合、この方法だと20枚以上の画像を用意しなければなりません。これだけでも大変ですが、画像を差し替えたり壁の種類を増やしたりしようとすればもっと大変になります。昔のコンピューターは実際にたくさんの画像を用意していたのですが、現代のコンピューターには画像の拡大縮小を行う機能が備わっています。この機能を使えば、少ない枚数の画像でも広い空間を描くことができるはずです。

// 壁のある部分に画像を配置.  
**+**const float base\_scales[check\_height] = // 奥行きに対する拡大率.  
**+** { 0.0625, 0.125, 0.25, 0.5, 1 };  
**+**const float wall\_x\_scales[check\_width] = // 側面の壁のX方向の拡大率.  
**+** { 3, 1, 0, -1, -3 };  
**+**const float wall\_front\_offsets[check\_width][2] = // 正面の壁の配置座標.  
**+** { { -2048, 0 },{ -1024, 0 },{ 0, 0 },{ 1024, 0 },{ 2048, 0 } };  
**+**const float wall\_side\_offsets[check\_width][2] = // 側面の壁の配置座標.  
**+** { { -1152, 0 },{ -384, 0 },{ 0, 0 },{ 384, 0 },{ 1152, 0 } };  
**+**const int x\_order[check\_width] = { 0, 4, 1, 3, 2 }; // X方向マスの配置順序.  
**+**  
**+**int image\_no = 1; // 次に配置する画像の管理番号.  
**+**set\_image(0, 0, 0, "dungeon\_background.png");  
**+**for (int b = 0; b < check\_height; ++b) {  
**+** const float offset\_scale = base\_scales[b];  
**+** const float image\_scale = base\_scales[b] \* 2; // 画像が小さいので2倍する.  
**+** for (int x = 0; x < check\_width; ++x) {  
**+** const int a = x\_order[x];  
**+** const int wall\_type = vicinity\_map[b][a];  
**+** if (wall\_type > 0) {  
**+** // 現在地より奥のマスなら正面の壁を描く.  
**+** if (b < check\_height - 1) {  
**+** set\_image(image\_no, 0, 0, "dungeon\_wall\_front.png");  
**+** move\_image(image\_no, wall\_front\_offsets[a][0] \* offset\_scale,  
**+** wall\_front\_offsets[a][1] \* offset\_scale, 0, 0);  
**+** scale\_image(image\_no, image\_scale, image\_scale, 0, 0);  
**+** ++image\_no;  
**+** }  
**+** // 正面以外のマスなら側面の壁を描く.  
**+** if (a != 2) {  
**+** set\_image(image\_no, 0, 0, "dungeon\_wall\_side.png");  
**+** move\_image(image\_no, wall\_side\_offsets[a][0] \* offset\_scale,  
**+** wall\_side\_offsets[a][1] \* offset\_scale, 0, 0);  
**+** scale\_image(image\_no, wall\_x\_scales[a] \* image\_scale,  
**+** image\_scale, 0, 0);  
**+** ++image\_no;  
**+** }  
**+** }  
**+** }  
**+**}

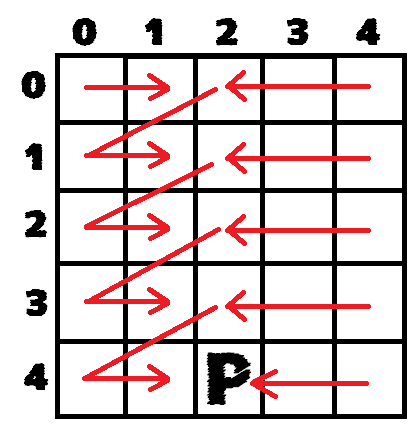
コピー&ペーストできそうな部分を見つけたら積極的に使っていってください(5割以上が同じプログラムになっている部分はコピペ対象と考えましょう。ほぼ確実に全部書くより早いです)。

まず最初に、いくつかの配列を設定しています。base\_scales(べーす・すけーるず)は奥の壁ほど小さく表示するための拡大率の表です。名称を複数形にしているのは、配列には複数の数値が入れられるからです。wall\_x\_scales(うぉーる・えっくす・すけーるず)は横に離れた壁ほど横長に表示するための拡大率です(垂直の壁を真横から見るとただの線ですが、横にずれていくに従って見える面積が増えていきますよね)。



wall\_front\_offsets(うぉーる・ふろんと・おふせっつ)とwall\_side\_offsets(うぉーる・さいど・おふせっつ)はそれぞれ正面向きの壁と横向きの壁の画像を配置する座標です。画面の座標系は中心が0で右がプラス方向なので、左端のマスが最小で中央が0、そして右端が最大の座標になっています。

x\_order(えっくす・おーだー)は横方向の壁の配置順を決める配列です。前回から引き続き「近くの画像に大きい管理番号を割り当てる」というルールを採用しているのですが、このルールは奥行方向だけでなく、左右方向にも当てはまります。ですから、vicinity\_mapを設定したときのように左から順番に処理されていくのでは都合が悪いのです。解決策として、x\_order配列を参照することで「0,1,2,3,4」ではなく「0,4,1,3,2]という順番で画像が設定されるようにしているのです。つまり、配列を「ある順序をもとに異なる順序を作り出す」ために使っているわけです。



image\_noは画像の管理番号を制御する変数です。前回は5つのマス全てに管理番号を決めていましたが、今回のように表示するマスが増えてくると、壁のないマスに管理番号を割り当てるのがもったいなくなってきます。そこで、画像を配置するたびにimage\_noに1を足していき、使っていない番号が出ないようにしています。

現在地と同じ横列の正面の壁は描きません。その位置はプレイヤーの後ろになるので絶対に見えないからです。同様に、現在地と同じ縦列の側面の壁も描きません。正面の壁に隠れて見えないからです。見えない部分を描かないようにすることは、このプログラムでは管理番号の消費をおさえるという意味もありますが、より重要なのはコンピューターに無用な処理をさせないということです。いくらコンピューターの性能が高くなったといっても限界はあります。より重要な仕事(ここでは画面にうつる画像の表示)に注力させることが大切なのです。

それから、プログラムで使用している画像は、いつものようにshareフォルダにあります。

## 正面の壁の判定

前進できるかどうかの判定も、vicinity\_mapを使うように変更しましょう。  
プレイヤーの操作を判定するプログラムを次のように変更してください。

const int player\_action =  
 select(-360, -40, 4, "前進", "右を向く", "左を向く", "後ろを向く");  
 if (player\_action == 0) {  
**!** if (vicinity\_map[3][2] == 1) {  
 // 前方に壁がある場合は進めない.

これで壁の判定は完了です。

## 以前のプログラムを削除する

早速実行、といきたいところですが、そのまえにwall\_front, wall\_right, wall\_leftを使っている古いプログラムを消しておきます。

まずは前回の壁の有無を取得するプログラムを削除してください。

vicinity\_map[b][a] = dungeon\_map[y][x];  
 }  
 }  
 }  
}  
**-**int wall\_left[2] = {};  
**-**int wall\_right[2] = {};  
**-**int wall\_front[1] = {};  
**-**const int wall\_check\_list[][5][2] = {  
**-** { { 0, -1 },{ -1, -1 },{ 0, 1 },{ -1, 1 },{ -1, 0 } },  
**-** { { -1, 0 },{ -1, 1 },{ 1, 0 },{ 1, 1 },{ 0, 1 } },  
**-** { { 0, 1 },{ 1, 1 },{ 0, -1 },{ 1, -1 },{ 1, 0 } },  
**-** { { 1, 0 },{ 1, -1 },{ -1, 0 },{ -1, -1 },{ 0, -1 } },  
**-**};  
**-**wall\_left[0] = dungeon\_map[player\_y +  
**-** wall\_check\_list[player\_direction][0][0]][player\_x +  
**-** wall\_check\_list[player\_direction][0][1]];  
**-**wall\_left[1] = dungeon\_map[player\_y +  
**-**  wall\_check\_list[player\_direction][1][0]][player\_x +  
**-** wall\_check\_list[player\_direction][1][1]];  
**-**wall\_right[0] = dungeon\_map[player\_y +  
**-** wall\_check\_list[player\_direction][2][0]][player\_x +  
**-**  wall\_check\_list[player\_direction][2][1]];  
**-**wall\_right[1] = dungeon\_map[player\_y +  
**-**  wall\_check\_list[player\_direction][3][0]][player\_x +  
**-**  wall\_check\_list[player\_direction][3][1]];  
**-**wall\_front[0] = dungeon\_map[player\_y +  
**-**  wall\_check\_list[player\_direction][4][0]][player\_x +  
**-**  wall\_check\_list[player\_direction][4][1]];

続いて、次の壁の画像を配置するプログラムを削除してください。

++image\_no;  
 }  
 }  
 }  
}  
**-**set\_image(0, 0, 0, "dungeon\_background.png");  
**-**if (wall\_left[1]) {  
**-** set\_image(1, 0, 0, "dungeon\_left\_wall\_1.png");  
**-**}  
**-**if (wall\_left[0]) {  
**-** set\_image(2, 0, 0, "dungeon\_left\_wall\_0.png");  
**-**}  
**-**if (wall\_right[1]) {  
**-** set\_image(3, 0, 0, "dungeon\_right\_wall\_1.png");  
**-**}  
**-**if (wall\_right[0]) {  
**-** set\_image(4, 0, 0, "dungeon\_right\_wall\_0.png");  
**-**}  
**-**if (wall\_front[0]) {  
**-** set\_image(5, 0, 0, "dungeon\_front\_wall\_0.png");  
**-**}

(せっかく書いてもらったのに申し訳ない…)

両方とも削除したら、プログラムを実行してみてください。  
うまくいけば、これまでよりずっと遠くまで迷路が見えるようになっているはずです。

# 天井と床、そして…

## 配列を追加する

次は、天井と床を描いてみましょう。  
まずは天井と床のための配列を追加します。壁を描くプログラムの配列を設定している部分に、次のプログラムを追加してください。

const float wall\_side\_offsets[check\_width][2] = // 側面の壁の配置座標.  
 { { -1152, 0 },{ -384, 0 },{ 0, 0 },{ 384, 0 },{ 1152, 0 } };  
const int x\_order[check\_width] = { 0, 4, 1, 3, 2 }; // X方向マスの配置順序.  
**+**const float ceil\_offsets[check\_width][2] = // 天井の配置座標.  
**+** { { -1536, 384 },{ -768, 384 },{ 0, 384 },{ 768, 384 },{ 1536, 384 } };  
**+**const float floor\_offsets[check\_width][2] = // 床の配置座標.  
**+** { { -1536, -384 },{ -768, -384 },{ 0, -384 },{ 768, -384 },{ 1536, -384 } };  
**+**const float ceil\_shears[check\_width] = { -4, -2, 0, 2, 4 }; // 天井の傾き.  
**+**const float floor\_shears[check\_width] = { 4, 2, 0, -2, -4 }; // 床の傾き.  
  
int image\_no = 1; // 次に配置する画像の管理番号.  
set\_image(0, 0, 0, "dungeon\_background.png");

ceil\_offsets(せいる・おふせっつ)とfloor\_offsets(ふろあ・おふせっつ)は天井の床の画像を配置する座標です。ceilは「天井」、floorは「床」という意味の英単語です。なお、壁や天井や床の座標が同じでないのは、配置する位置が異なる、ということに加えて、画像の大きさも異なっているためです(正面の壁は正方形、側面の壁は縦長、床や天井は横長)。

ceil\_shears(せいる・しあーず)とfloor\_shears(ふろあ・しあーず)は、画像を傾ける(グラフィックスの用語で「剪断(せんだん)」といいます)比率を設定する配列です。これは新しく追加されたshear\_image関数用のパラメーターです。

## 天井と床を描く

続いて、天井と床の画像を配置します。壁の画像を配置するプログラムの手前に、次のプログラムを追加してください。

const int a = x\_order[x];  
 const int wall\_type = vicinity\_map[b][a];  
**+  
+**// 天井を表示. **+**set\_image(image\_no, 0, 0, "dungeon\_ceil.png");  
**+**move\_image(image\_no, ceil\_offsets[a][0] \* offset\_scale,  
**+** ceil\_offsets[a][1] \* offset\_scale, 0, 0);  
**+**scale\_image(image\_no, image\_scale, image\_scale, 0, 0);  
**+**shear\_image(image\_no, ceil\_shears[a], 0, 0);  
**+**++image\_no;  
**+**  
**+**// 床を表示. **+**set\_image(image\_no, 0, 0, "dungeon\_floor.png");  
**+**move\_image(image\_no, floor\_offsets[a][0] \* offset\_scale,  
**+** floor\_offsets[a][1] \* offset\_scale, 0, 0);  
**+**scale\_image(image\_no, image\_scale, image\_scale, 0, 0);  
**+**shear\_image(image\_no, floor\_shears[a], 0, 0);  
**+**++image\_no;

天井の表示も床の表示も、壁の表示と同じようなプログラムになっています。唯一の違いはshear\_image関数を使っていることです。この関数はCommand.hで宣言されていて、その内容は次のようなものです。

/\*\*  
\* 画像を傾ける.  
\*  
\* @param no 画像の管理番号.  
\* @param scale 傾ける比率. マイナス値を設定すると傾きが逆になる.  
\* @param easing 補間方法:  
\* 0 等速で傾く.  
\* 1 加速しながら傾く.  
\* 2 減速しながら傾く.  
\* 3 加速しながら傾き始めて、減速しながら停止する.  
\* 4 1に似ているが、少し傾き過ぎてから戻ってきて停止する.  
\* 5 1に似ているが、何度か弾みなから停止する.  
\* @param seconds 動作時間(秒).  
\*/  
void shear\_image(int no, float scale, int easing, float seconds);

引数はmove\_imageやrotate\_imageなどとほぼ同じです。この関数は、画像を次のように変形します。



床や天井は左右に離れるほど変形して見えるので、この関数を使って変形させるわけです。

プログラムが書けたら実行してみてください。  
天井と床が表示されたでしょうか。

## ダンジョンマップに扉を追加する

どんなによくできた迷路だったとしても、壁だけが延々と続くのでは飽きがきてしまいます。そこで、扉を追加してみましょう。マップデータにおいて、扉には32番を割り当てることにします。31番までは壁の種類を増やしたくなったときのために予約しておきます。

ダンジョンの大きさを設定しているプログラムの下に、次のプログラムを追加してください。

// ダンジョンの大きさ  
 const int dungeon\_width = 8;  
 const int dungeon\_height = 8;  
**+**  
**+**// ダンジョンマップにおいて扉であることを示す番号.  
**+**const int wall\_type\_door = 32;

wall\_type\_door(うぉーる・たいぷ・どあ)は、壁ではなく扉であることを示す番号です。

次に、ダンジョンマップに扉を追加しましょう。  
ダンジョンマップを次のように変更してください。

const char dungeon\_map[dungeon\_width][dungeon\_height] = {  
 { 1, 1, 1, 1, 1, 1, 1, 1 },  
**!**{ 1, 0, 1, 0,32, 0, 0, 1 },  
 { 1, 0, 1, 1, 1, 0, 1, 1 },  
 { 1, 0, 0, 0, 1, 0, 0, 1 },  
**!**{ 1, 0, 1,32, 1, 1,32, 1 },  
**!**{ 1, 1, 1, 0,32, 0, 0, 1 },  
 { 1, 0, 0, 0, 1, 0, 1, 1 },  
 { 1, 1, 1, 1, 1, 1, 1, 1 },  
 };

通路だった部分の何箇所かに扉を設置しました。

さて、扉は壁ではないので前進できるようにしたいですね。  
前方が壁かどうかを判定しているプログラムを、次のように変更してください。

if (player\_action == 0) {  
**!** if (vicinity\_map[3][2] > 0 && vicinity\_map[3][2] < wall\_type\_door) {  
 // 前方に壁がある場合は進めない.

0より大きく32より小さい場合は壁だと判定するようにしてみました。

## 扉を表示する

マップの情報として扉を配置しただけでは画面には表示されません。プログラムで書いていないのだから当然ですね。そこで、扉を表示するプログラムを追加しましょう。  
壁を表示するプログラムに、次のプログラムを追加してください。

if (b < check\_height - 1) {  
 set\_image(image\_no, 0, 0, "dungeon\_wall\_front.png");  
 move\_image(image\_no, wall\_front\_offsets[a][0] \* offset\_scale,  
 wall\_front\_offsets[a][1] \* offset\_scale, 0, 0);  
 scale\_image(image\_no, image\_scale, image\_scale, 0, 0);  
 ++image\_no;  
**+** // 壁の種類が「扉」なら扉の画像も配置する.  
**+** if (wall\_type >= wall\_type\_door) {  
**+** set\_image(image\_no, 0, 0, "dungeon\_door\_front.png");  
**+** move\_image(image\_no, wall\_front\_offsets[a][0] \* offset\_scale,  
**+** wall\_front\_offsets[a][1] \* offset\_scale, 0, 0);  
**+** scale\_image(image\_no, image\_scale, image\_scale, 0, 0);  
**+** ++image\_no;  
**+** }  
 }  
 if (a != 2) {  
 set\_image(image\_no, 0, 0, "dungeon\_wall\_side.png");  
 move\_image(image\_no, wall\_side\_offsets[a][0] \* offset\_scale,  
 wall\_side\_offsets[a][1] \* offset\_scale, 0, 0);  
 scale\_image(image\_no, wall\_x\_scales[a] \* image\_scale, image\_scale, 0, 0);  
 ++image\_no;  
**+** // 壁の種類が「扉」なら扉の画像も配置する.  
**+** if (wall\_type >= wall\_type\_door) {  
**+** set\_image(image\_no, 0, 0, "dungeon\_door\_side.png");  
**+** move\_image(image\_no, wall\_side\_offsets[a][0] \* offset\_scale,  
**+** wall\_side\_offsets[a][1] \* offset\_scale, 0, 0);  
**+** scale\_image(image\_no, wall\_x\_scales[a] \* image\_scale, image\_scale, 0, 0);  
**+** ++image\_no;  
**+** }  
 }

壁と同様に扉にも正面と側面で異なる画像があるので2箇所にプログラムを追加しています。追加したプログラムですが、実は直前の壁を配置するプログラムとほとんどおなじです(ファイル名が違うだけ)。というのも、壁と扉を同じ大きさの画像にしてあるからです。データの大きさを統一しておけば、コピー&ペーストがはかどるということです。

プログラムが書けたら実行してみてください。  
扉は表示されたでしょうか。

## ダンジョンマップに泉を追加する

扉以外にも何か追加しましょう。そうですね…「泉」なんてどうでしょう？  
都合のいいことに、泉の画像はもう用意してあるんです。

まずは扉のときと同様に、泉に割り当てる番号を決めます。今回用意した泉の画像は、床の上に配置することを意図しています。現在のプログラムでは1以上の番号を「壁」の一種として扱っていますので、床に配置するものは1未満の番号、つまりマイナスの値にするべきでしょう。扉のときと同様に、床のバリエーションを増やせるようにもしておきたいですね。ということで、泉の番号は-32番とします。  
今後の番号追加に備えて、コメントもわかりやすく修正しておくことにします。

それではプログラムしていきましょう。  
扉の番号を設定しているプログラム付近を、次のように変更してください。

// ダンジョンの大きさ  
const int dungeon\_width = 8;  
const int dungeon\_height = 8;  
  
**!**// ダンジョンマップの地形番号.  
**!**const int wall\_type\_door = 32; // 扉.  
**+**const int wall\_type\_fountain = -32; // 泉.

wall\_type\_fountain(うぉーる・たいぷ・ふぁうんてん)が泉の番号を示す変数です。fountainは「泉」や「噴水」という意味の英単語です。  
ダンジョンの地形を追加したくなったら、泉の下に追加することができます。

次に、ダンジョンマップに泉を追加します。  
ダンジョンマップを次のように変更してください。

const char dungeon\_map[dungeon\_width][dungeon\_height] = {  
 { 1, 1, 1, 1, 1, 1, 1, 1 },  
 { 1, 0, 1, 0,32, 0, 0, 1 },  
 { 1, 0, 1, 1, 1, 0, 1, 1 },  
 { 1, 0, 0, 0, 1, 0, 0, 1 },  
**!**{ 1,-32,1,32, 1, 1,32, 1 },  
 { 1, 1, 1, 0,32, 0, 0, 1 },  
 { 1, 0, 0, 0, 1, 0, 1, 1 },  
 { 1, 1, 1, 1, 1, 1, 1, 1 },  
 };

最初の通路の行き止まりに泉をひとつ配置してみました。

## 泉を表示する

泉を表示するプログラムを追加します。  
まずは泉の表示位置を配列として追加します。  
壁の表示位置の配列の下に、次のプログラムを追加してください。

const float wall\_side\_offsets[check\_width][2] = // 側面の壁の配置座標.  
 { { -1152, 0 },{ -384, 0 },{ 0, 0 },{ 384, 0 },{ 1152, 0 } };  
**+**const float floor\_object\_offsets[check\_width][2] = // フロア内の物体の配置座標.  
**+** { { -1536, 0 },{ -768, 0 },{ 0, 0 },{ 768, 0 },{ 1536, 0 } };  
 const int x\_order[check\_width] = { 0, 1, 4, 3, 2 };  
 int image\_no = 1;

続いて、床を配置するプログラムの下に、次のプログラムを追加してください。

shear\_image(image\_no, floor\_shears[a], 0, 0);  
 ++image\_no;  
**+  
+**if (wall\_type == wall\_type\_fountain) {  
**+** // 泉を表示.  
**+** set\_image(image\_no, 0, 0, "dungeon\_fountain.png");  
**+** move\_image(image\_no, floor\_object\_offsets[a][0] \* offset\_scale,  
**+** floor\_object\_offsets[a][1] \* offset\_scale, 0, 0);  
**+** scale\_image(image\_no, image\_scale, image\_scale, 0, 0);  
**+** ++image\_no;  
**!**} else if (wall\_type > 0) {  
 if (b < check\_height - 1) {

泉は円形で、同じ画像をどの方向から見てもあまり違和感を感じないようにしています。

プログラムが書けたら、実行してみてください。  
行く手に泉(のようなもの)が表示され、近づいて行くことができればうまくいっています。

# 扉に鍵をかける

## 鍵のかかった扉と鍵の位置を設定する

迷宮というからには謎のひとつも必要でしょう。  
そこで、出口の手前の扉に鍵をかけてしまおうと思います。  
ゴール位置を設定しているプログラムの下に、次のプログラムを追加してください。

// ゴールの位置.  
 const int goal\_x = 3;  
 const int goal\_y = 1;  
**+**  
**+**// 鍵のかかった扉の位置.  
**+**const int locked\_door\_x = 4;  
**+**const int locked\_door\_y = 1;  
**+**  
**+**// 鍵の位置.  
**+**const int key\_x = 1;  
**+**const int key\_y = 6;  
**+**  
**+**// 鍵を持っているか.  
**+**bool has\_key;  
  
 // 戦闘中のときはtrue. そうでなければfalse.  
 bool battle\_flag;

次に、ゲームの初期設定を行うプログラムに、次のプログラムを追加してください。

// ゲームの初期設定を行う.  
 player\_x = start\_x;  
 player\_y = start\_y;  
 player\_direction = dir\_down;  
 player\_hp = player\_hp\_max;  
 aiko\_flag = false;  
**+**has\_key = false;  
 battle\_flag = false;

ゲームを開始する時に鍵を未所持の状態にしています。

## 鍵のかかった扉の処理

特定の位置で扉に前進しようとしたとき、鍵を所持しているかどうかによって処理を分けます。  
プレイヤーを前進させるプログラムを次のように変更してください。

} else {  
**+** const int move[4][2] = { { 0, -1 },{ 1, 0 },{ 0, 1 },{ -1, 0 } };  
**+** const int front\_x = player\_x + move[player\_direction][0];  
**+** const int front\_y = player\_y + move[player\_direction][1];  
**+** bool is\_open\_front = true; // 前進できるならtrue. できなければfalse.  
**+** // 鍵のかかった扉の処理.  
**+** if (front\_x == locked\_door\_x && front\_y == locked\_door\_y) {  
**+** reset\_text\_area(-400, -300, 800, 301);  
**+** if (has\_key) {  
**+** set\_text(-360, 0, "鍵を開けた.");  
**+** } else {  
**+** set\_text(-360, 0, "鍵がかかっている.");  
**+** is\_open\_front = false;  
**+** }  
**+** wait(1.5);  
**+** }  
 // 壁がなければ、プレイヤーの前方へ移動する.  
**+** if (is\_open\_front) {  
**-** const int move[4][2] = { { 0, -1 },{ 1, 0 },{ 0, 1 },{ -1, 0 } };  
 player\_x += move[player\_direction][0];  
 player\_y += move[player\_direction][1];  
**+** }

プレイヤーの前方が鍵のかかった扉の座標だった場合、鍵を所持している場合のみ前進できるようにしています。また、所持状態によって表示するメッセージを切り替えています。

## 鍵を拾う処理

最後に鍵を拾う処理を追加しましょう。  
プレイヤーの行動を選択する処理の手前に、次のプログラムを追加してください。

**+**// 鍵を拾う処理.  
**+**if (!has\_key && player\_x == key\_x && player\_y == key\_y) {  
**+** set\_text(-360, 0, "鍵が落ちている.");  
**+** wait(1);  
**+** set\_text(-360, -40, "鍵を拾いました.");  
**+** set\_text(-360, -80, "(何かキーを押してください)");  
**+** has\_key = true;  
**+** wait\_any\_key();  
**+** reset\_text\_area(-400, -300, 800, 301);  
**+**}  
**+**  
 // ゴールではないので行動を選択する.  
 set\_text(-360, 0, "どうしますか？");  
 const int player\_action =  
 　　select(-360, -40, 4, "前進", "右を向く", "左を向く", "後ろを向く");

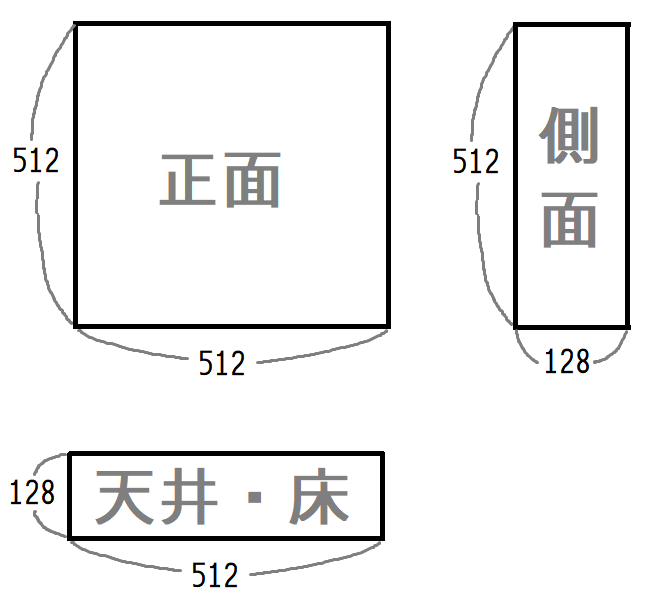
プレイヤーが鍵を持っていない状態で鍵の座標にきたら、鍵を取得させています。

プログラムが書けたら実行してみてください。  
最後の扉が鍵がないと開けられなくなっていれば成功です。

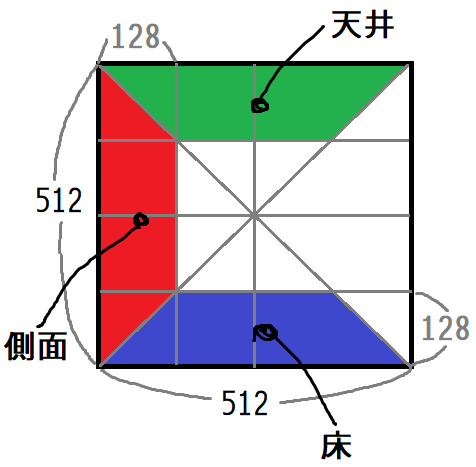
[課題]  
1. 扉や泉の位置を変更してください。  
 鍵のかかった扉の位置を変更するときは、locked\_x,locked\_yも変更することを忘れないように。  
2. プレイヤーが泉のマスに着いたとき、何らかのメッセージを表示してください。  
3. 迷宮に鍵以外のアイテムを配置して、入手できるようにしてください。  
 どんなアイテムなのかはお任せします(第2の鍵とかでも構いません)。

# 迷宮の画像について

画像を差し替える、あるいは追加する場合、下記の図のサイズのものを用意してください。  
扉の画像は正面及び側面の壁に準じます。泉の画像は正面の壁に準じます。



側面の壁、天井、床については、512x512サイズの画像を、次のような形状に変形させたものになっています。



このような変形を行うには、GIMPのperspective(遠近法)ツールなどを使うのが簡単です。  
GIMPの遠近法ツールは台形のアイコンになっています。

