ブロックの世界を散歩

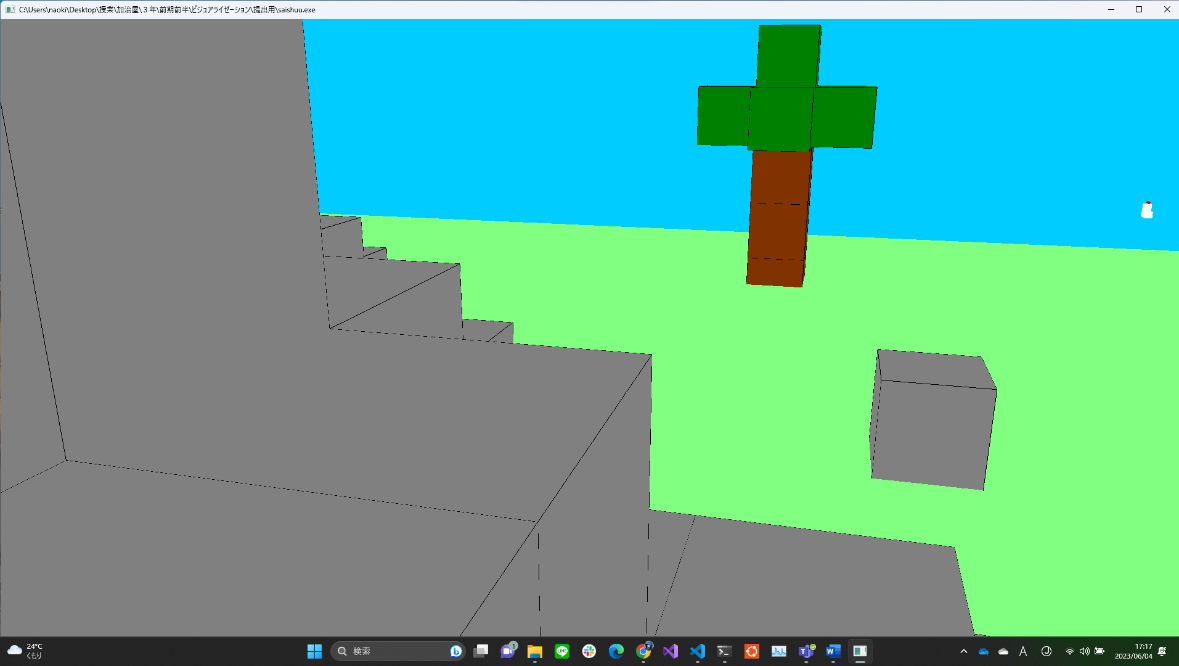
（プログラム設計書）

情報システム学科　加治屋直希

申請点１００点

１.ソフトウェア概要

本ソフトウェアは、ブロックでできた世界を一人称視点で散歩するものである。以下が実際のプレイ中の画像である。



2.関数一覧



３．変数一覧（グローバル変数のみ）



４．モデリング

　モデリングに関するスクリプトは主に、５つの関数として書かれている。それぞれの関数の概要は[関数一覧](#関数一覧)を参考にしてほしい。また、グローバル変数については[変数一覧](#変数一覧)を参照してほしい。

1. Ground

**void Ground(void)**

**{**

**glPushMatrix();**

**glBegin(GL\_QUADS);**

**glColor3f(0.5, 1.0, 0.5);**

**glVertex3f(nowx - ground\_max\_x - 10.0, nowy + ground\_max\_y + 10.0, 0.);**

**glVertex3f(nowx + ground\_max\_x + 10.0, nowy + ground\_max\_y + 10.0, 0.);**

**glVertex3f(nowx + ground\_max\_x + 10.0, nowy - ground\_max\_y - 10.0, 0.);**

**glVertex3f(nowx - ground\_max\_x - 10.0, nowy - ground\_max\_y - 10.0, 0.);**

**glEnd();**

**glPopMatrix();**

**}**

１つ目の関数はGround関数である。この関数では地面を現在地の周り45

×45ブロック分を[glVertex3f]で設定し、[GL\_QUADS]で作成している。そのため、移動しても、地面のみならば無限に描画できる。例えば現在地が左の図の赤い部分だとすると、左右上下それぞれ22マス離れたところまで（緑色の部分）が描画されるようになっている。

1. ｂrock

**void brock(float r, float g, float b, float x, float y, float z)**

**{**

**glPushMatrix();**

**glTranslatef(x, y, z);**

**glColor3f(0.0, 0.0, 0.0);**

**glutWireCube(1.00);**

**glColor3f(r, g, b);**

**glEnable(GL\_NORMALIZE);**

**glBegin(GL\_QUADS);**

**glTexCoord2f(0.0, 0.0);**

**glVertex3f(-0.5, -0.5, -0.5);**

**glTexCoord2f(0.3, 0.0);**

**glVertex3f(-0.5, 0.5, -0.5);**

**glTexCoord2f(0.3, 0.5);**

**glVertex3f(0.5, 0.5, -0.5);**

**glTexCoord2f(0.0, 0.5);**

**glVertex3f(0.5, -0.5, -0.5);**

**glEnd();**

**glBegin(GL\_QUADS);**

**glTexCoord2f(0.7, 0.5);**

**glVertex3f(-0.5, -0.5, 0.5);**

**glTexCoord2f(1.0, 0.5);**

**glVertex3f(-0.5, 0.5, 0.5);**

**glTexCoord2f(1.0, 1.0);**

**glVertex3f(0.5, 0.5, 0.5);**

**glTexCoord2f(0.7, 1.0);**

**glVertex3f(0.5, -0.5, 0.5);**

**glEnd();**

**glBegin(GL\_QUADS);**

**glTexCoord2f(0.4, 0.0);**

**glVertex3f(-0.5, -0.5, -0.5);**

**glTexCoord2f(0.7, 0.0);**

**glVertex3f(-0.5, -0.5, 0.5);**

**glTexCoord2f(0.7, 0.5);**

**glVertex3f(0.5, -0.5, 0.5);**

**glTexCoord2f(0.4, 0.5);**

**glVertex3f(0.5, -0.5, -0.5);**

**glEnd();**

**glBegin(GL\_QUADS);**

**glTexCoord2f(0.4, 0.5);**

**glVertex3f(-0.5, 0.5, -0.5);**

**glTexCoord2f(0.7, 0.5);**

**glVertex3f(-0.5, 0.5, 0.5);**

**glTexCoord2f(0.7, 1.0);**

**glVertex3f(0.5, 0.5, 0.5);**

**glTexCoord2f(0.4, 1.0);**

**glVertex3f(0.5, 0.5, -0.5);**

**glEnd();**

**glBegin(GL\_QUADS);**

**glTexCoord2f(0.7, 0.0);**

**glVertex3f(0.5, -0.5, -0.5);**

**glTexCoord2f(1.0, 0.0);**

**glVertex3f(0.5, 0.5, -0.5);**

**glTexCoord2f(1.0, 0.5);**

**glVertex3f(0.5, 0.5, 0.5);**

**glTexCoord2f(0.7, 0.5);**

**glVertex3f(0.5, -0.5, 0.5);**

**glEnd();**

**glBegin(GL\_QUADS);**

**glTexCoord2f(0.0, 0.5);**

**glVertex3f(-0.5, -0.5, -0.5);**

**glTexCoord2f(0.35, 0.5);**

**glVertex3f(-0.5, 0.5, -0.5);**

**glTexCoord2f(0.35, 1.0);**

**glVertex3f(-0.5, 0.5, 0.5);**

**glTexCoord2f(0.0, 1.0);**

**glVertex3f(-0.5, -0.5, 0.5);**

**glEnd();**

**glDisable(GL\_NORMALIZE);**

**glPopMatrix();**

**}**

２つ目の関数はbrock関数である。この関数では３次元ベクトルを[glVertex3f]で設定し、そこに[GL\_QUADS]で面を作成し、６回それぞれ繰り返すことで６面の立方体を生成している。引数にはRGB値のための[r/g/b]と座標上のどこに生成するか指定するための[x/y/z]が必要である。

1. tree

**void tree(float x, float y, float z)**

**{**

**glPushMatrix();**

**int bx, by;**

**/\*幹\*/**

**for (float i = 0.; i < 3.; i += 1.0)**

**{**

**brock(0.5, 0.2, 0.0, x, y, z + i);**

**}**

**/\*葉\*/**

**brock(0.0, 0.5, 0.0, x - 1.0, y, z + 3.0);**

**brock(0.0, 0.5, 0.0, x + 1.0, y, z + 3.0);**

**brock(0.0, 0.5, 0.0, x, y - 1.0, z + 3.0);**

**brock(0.0, 0.5, 0.0, x, y + 1.0, z + 3.0);**

**brock(0.0, 0.5, 0.0, x, y, z + 4.0);**

**bx = changeToBrock(x);**

**by = changeToBrock(y);**

**ground[bx][by] = 4.0;**

**glPopMatrix();**

**}**

　３つ目の関数は、tree関数である。この関数は木を生成するために用いる。

左の図が木のイメージである。左から上から見た、下から見た、横から見た状態である。関数内では、brock関数を用いて木のモデリングをし、changeToBrock関数で座標をground配列のインデックスに変換し、木の座標部分の地形情報に高さ４にしている。changeToBrock関数については後で詳しく説明する。引数には視点座標系における生成する場所の座標が必要である。

1. rockMountain

**void rockMountain(float theta, float x, float y)**

**{**

**glPushMatrix();**

**glRotated(theta, 0.0, 0.0, 1.0);**

**glTranslatef(x, y, 0.0);**

**brock(0.5, 0.5, 0.5, -3.0, 10.0, 0.5);**

**.**

**（中略）**

**.**

**brock(0.5, 0.5, 0.5, -10.0, 6.0, 3.5);**

**glPopMatrix();**

**}**

４つ目の関数は、rockMountain関数である。基本的にtree関数と同じであり、brock関数を用いて任意の形の岩山を生成している。引数に座標とthetaというfloaｔ型の変数を持っている。座標は視点座標系における生成したい場所を示し、thetaには回転角を入れる。角度を用いることによって、原点対称の岩山に関しては、一部スクリプトを省略することが可能となる。

たとえば、左下の図のような地形をrockMountain関数で生成し、thetaの引数にそれぞれ、｛0/90/180/270｝を代入し、それぞれに対応した座標を代入すれば、岩山を生成できる。その結果が右下の図である。



1. mob

**/\*くちばし\*/**

**glPushMatrix();**

**glTranslatef(-0.75, 0.0, 1.5);**

**glColor3f(1.0, 1.0, 0.0);**

**glScalef(1.0 / 3.0, 1.0 / 3.0, 1.0 / 3.0);**

**glutSolidCube(1.0);**

**glPopMatrix();**

**/\*しっぽ\*/**

**glPushMatrix();**

**glTranslatef(1.75, 0.0, 0.5);**

**glColor3f(1.0, 1.0, 1.0);**

**glScalef(1.0 / 3.0, 1.0 / 3.0, 1.0 / 3.0);**

**glutSolidCube(1.0);**

**glPopMatrix();**

**glDisable(GL\_NORMALIZE);**

**}**

**void mob(void)**

**{**

**glEnable(GL\_NORMALIZE);**

**/\*頭\*/**

**glPushMatrix();**

**glTranslatef(0.0, 0.0, 1.5);**

**glColor3f(1.0, 1.0, 1.0);**

**glutSolidCube(1.0);**

**glPopMatrix();**

**/\*胴\*/**

**glPushMatrix();**

**glTranslatef(0.5, 0.0, 0.5);**

**glColor3f(1.0, 1.0, 1.0);**

**glScalef(2.0, 1.0, 1.0);**

**glutSolidCube(1.0);**

**glPopMatrix();**

**/\*とさか\*/**

**glPushMatrix();**

**glTranslatef(0.25, 0.0, 2.25);**

**glColor3f(1.0, 0.0, 0.0);**

**glScalef(2.0 / 3.0, 1.0 / 3.0, 1.0 / 3.0);**

**glutSolidCube(1.0);**

**glPopMatrix();**

最後の関数は、mob関数である。この関数では、brock関数ではなく[glutSolidCube]を用いている。これは、[ｇｌScalef]によって、ブロックを縮めたり、引き延ばすとき、関数より扱いやすいと考えたためである。このmobはニワトリをイメージしたものになっている。下の図がイメージである。左から、上から見た、下から見た、横から見た状態である。

　また、display関数内でflagの値により、くちばしの方向をx軸正方向かx軸負方向化を決め、[glutRotatef]で向きを変えている。

５．対話性

　対話性に関するスクリプトは、４つの関数として書かれている。モデリングと同様に、概要は[関数一覧](#関数一覧)、[変数一覧](#変数一覧)を参照してほしい。

1. mykbd

**case 'd':**

**nowx += 1.0 \* cos((thetaXY - 90) \* M\_PI / 180);**

**lookfromx += 1.0 \* cos((thetaXY - 90) \* M\_PI / 180);**

**lookatx += 1.0 \* cos((thetaXY - 90) \* M\_PI / 180);**

**nowy += 1.0 \* sin((thetaXY - 90) \* M\_PI / 180);**

**lookfromy += 1.0 \* sin((thetaXY - 90) \* M\_PI / 180);**

**lookaty += 1.0 \* sin((thetaXY - 90) \* M\_PI / 180);**

**break;**

**case ' ':**

**nowz = ground[x][y] + 1;**

**setZ(nowz);**

**jump = TRUE;**

**break;**

**case KEY\_ESC:**

**exit(0);**

**break;**

**default:**

**break;**

**}**

**printf("mykd\n");**

**printf("lookat(%f,%f, %f) lookfrom(%f, %f, %f) thetaXY:%f\n", lookatx, lookaty, lookatz, lookfromx, lookfromy, lookfromz, thetaXY);**

**defaultValue();**

**glutPostRedisplay();**

**}**

**void mykbd(unsigned char key, int x, int y)**

**{**

**x = changeToBrock(nowx);**

**y = changeToBrock(nowy);**

**if (judge(key) == FALSE)**

**return;**

**switch (key)**

**{**

**case 'w':**

**nowx += 1.0 \* cos(thetaXY \* M\_PI / 180);**

**lookfromx += 1.0 \* cos(thetaXY \* M\_PI / 180);**

**lookatx += 1.0 \* cos(thetaXY \* M\_PI / 180);**

**nowy += 1.0 \* sin(thetaXY \* M\_PI / 180);**

**lookfromy += 1.0 \* sin(thetaXY \* M\_PI / 180);**

**lookaty += 1.0 \* sin(thetaXY \* M\_PI / 180);**

**break;**

**case 's':**

**nowx += 1.0 \* cos((thetaXY + 180) \* M\_PI / 180);**

**lookfromx += 1.0 \* cos((thetaXY + 180) \* M\_PI / 180);**

**lookatx += 1.0 \* cos((thetaXY + 180) \* M\_PI / 180);**

**nowy += 1.0 \* sin((thetaXY + 180) \* M\_PI / 180);**

**lookfromy += 1.0 \* sin((thetaXY + 180) \* M\_PI / 180);**

**lookaty += 1.0 \* sin((thetaXY + 180) \* M\_PI / 180);**

**break;**

**case 'a':**

**nowx += 1.0 \* cos((thetaXY + 90) \* M\_PI / 180);**

**lookfromx += 1.0 \* cos((thetaXY + 90) \* M\_PI / 180);**

**lookatx += 1.0 \* cos((thetaXY + 90) \* M\_PI / 180);**

**nowy += 1.0 \* sin((thetaXY - 90) \* M\_PI / 180);**

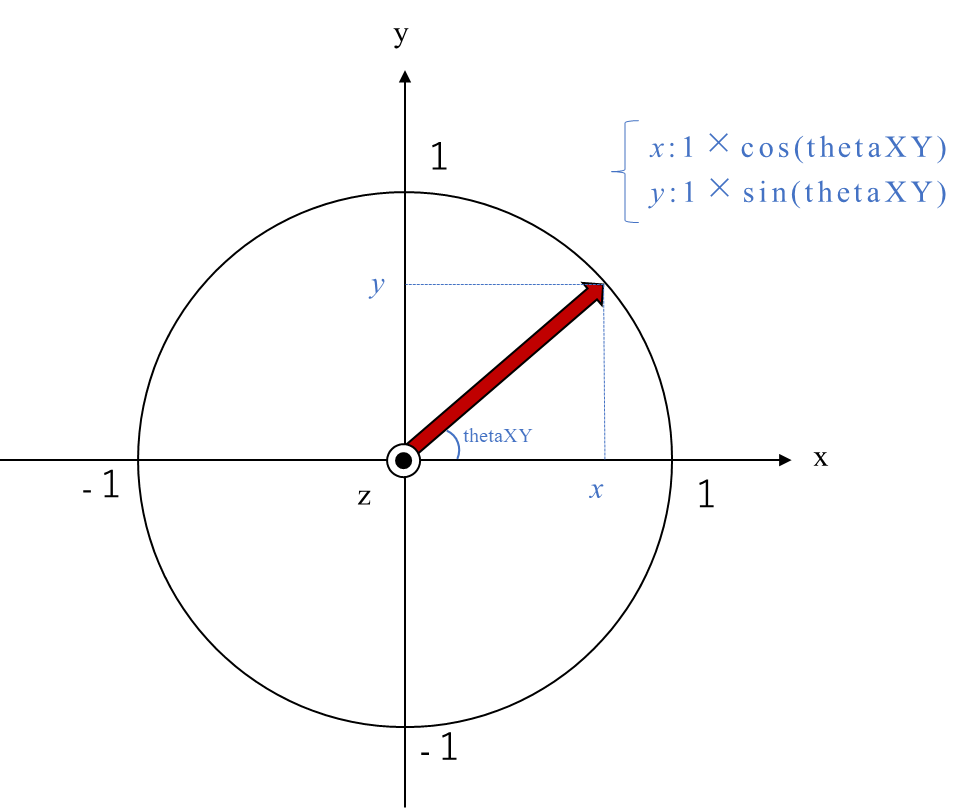
**lookfromy += 1.0 \* sin((thetaXY + 90) \* M\_PI / 180);**

**lookaty += 1.0 \* sin((thetaXY + 90) \* M\_PI / 180);**

**break;**

１つ目の関数は、mykbd関数である。この関数では、キーが入力された瞬間に視点移動を行っている。具体的には、[w/s/a/d]キーで前後、左右に移動し、ス[ペース]キーでジャンプしている。また[Esc]キーでシステムが終了する。x軸正方向に対する視線の傾きをthetaXYとし、それぞれの方向へ進んだ後の座標を余弦、正弦を用いて計算し、その値を視点座標と現在地座標に代入している。

計算に関する考え方は下図を参照してほしい。なお、赤矢印が視点の向きを表している。実際には、原点座標はnowx,nowyまたはlookfromx,lookfromyとなるので、それに計算した視点の座標の変化量を足すことで視点の座標を求める。



1. myupkbd

**void myupkbd(unsigned char key, int x, int y)**

**{**

**x = changeToBrock(nowx);**

**y = changeToBrock(nowy);**

**switch (key)**

**{**

**case ' ':**

**jump = FALSE;**

**nowz = ground[x][y];**

**setZ(nowz);**

**break;**

**default:**

**break;**

**}**

**glutPostRedisplay();**

**}**

２つ目の関数は、myupkbd関数である。この関数では、ジャンプ後の視点移動をキー入力が終わった瞬間に行っている。mykbd関数で[スペース]キーを押したときに、視点のz軸を+１し、myupkbd関数で-1しているこれにより、ジャンプしているように見せる。

1. viewPointkdDown

**lookatx = lookfromx + 0.5 \* cos(thetaXY \* M\_PI / 180);**

**lookaty = lookfromy + 0.5 \* sin(thetaXY \* M\_PI / 180);**

**printf("lookat(%f,%f,%f) lookfrom(%f, %f, %f) thetaXY:%f\n", lookatx, lookaty, lookatz, lookfromx, lookfromy, lookfromz, thetaXY);**

**glutPostRedisplay();**

**}**

**void viewPointkdDown(int key, int x, int y)**

**{**

**switch (key)**

**{**

**case GLUT\_KEY\_UP:**

**lookatz = lookfromz + 0.1;**

**break;**

**case GLUT\_KEY\_DOWN:**

**lookatz = lookfromz - 0.3;**

**break;**

**case GLUT\_KEY\_LEFT:**

**thetaXY += 15;**

**break;**

**case GLUT\_KEY\_RIGHT:**

**thetaXY -= 15;**

**break;**

**default:**

**break;**

**}**

３つ目の関数は、viewPointkdDownである。この関数では、視線移動をキーが入力された瞬間に行っている。[↑/↓]キーが入力された場合は、それぞれ視線のｚ座標を+0.1,-0.3変化されている。これらは固定の値となているため、押し続けて加算されることはない。また、[←/→]キーでthetaXYを+15,-15ずつ変化させている。視点移動と同様の考え方で計算し、現在の視点の座標（計算時の原点）に計算した視線の座標の変化量を足すことで視線の座標をを計算している。

1. viewPointkdUp

**void viewPointkdUp(int key, int x, int y)**

**{**

**switch (key)**

**{**

**case GLUT\_KEY\_UP:**

**lookatz = lookfromz - 0.15;**

**break;**

**case GLUT\_KEY\_DOWN:**

**lookatz = lookfromz - 0.15;**

**break;**

**default:**

**break;**

**}**

**glutPostRedisplay();**

**}**

最後の関数は、viewPointkdUp関数である。この関数は、上下の視線移動をキーが離れた瞬間に行っている。具体的には、それぞれの視線のz座標を初期値に戻している。

６．アニメーション

　アニメーションはidle関数にて行っている。概要は、[関数一覧](#関数一覧)・[変数一覧](#変数一覧)を参照。

**void idle(void)**

**{**

**int x = changeToBrock(nowx);**

**int y = changeToBrock(nowy);**

**if (jump == FALSE)**

**{**

**setZ(ground[x][y]);**

**}**

**if (flag == 0)**

**{**

**mobx += 0.001;**

**if (mobx >= 50)**

**{**

**flag = 1;**

**}**

**}**

**else**

**{**

**mobx -= 0.001;**

**if (mobx <= -50)**

**{**

**flag = 0;**

**}**

**}**

**glutPostRedisplay();**

**}**

　　色が濃くついている部分がアニメーションに関するスクリプトである。flagが0の時はx軸正方向に、flagが1の時はｘ軸負方向にそれぞれmobが0.001ずつ移動している。また、このプログラムでは、マップの広さを101×１０１に設定しているので、端まで移動したら折り返すようなプログラムになっている。

７．アルゴリズム

　そのほかの関数について説明する。概要は、[関数一覧](#関数一覧)・[変数一覧](#変数一覧)を参照。

1. defaultValue

**void defaltValue(void)**

**{**

**FILE \*fp;**

**int x = changeToBrock(nowx);**

**int y = changeToBrock(nowy);**

**printf("(%d, %d)\n", x, y);**

**fp = fopen("ground.txt", "r");**

**if (fp == NULL)**

**exit(1);**

**for (int i = 0; i < 101; i++)**

**{**

**for (int j = 0; j < 101; j++)**

**{**

**ground[i][j] = fgetc(fp) - 48;**

**if (ground[i][j] == EOF)**

**break;**

１つ目の関数は、defaultValue関数である。この関数では、ground配列にそれぞれのx,y座標に対応するz座標の情報を代入している。ｚ座標の情報はsaishuu.cと同フォルダに存在するground.txtに書かれており、[fopen/fgetc/fclose]によってground配列に代入している。また、[fprintf]によって書き出しも行うので、プログラムに書き込むだけで簡単にｚ軸情報を修正できる。

**if (ground[i][j] == -38)**

**{**

**/\*改行の時\*/**

**ground[i][j] = fgetc(fp) - 48;**

**if (ground[i][j] == EOF)**

**break;**

**}**

**}**

**}**

**fclose(fp);**

**fp = fopen("ground\_1.txt", "w");**

**if (fp == NULL)**

**exit(1);**

**for (int i = 0; i < 101; i++)**

**{**

**for (int j = 0; j <= 101; j++)**

**{**

**if (j == 101)**

**{**

**fprintf(fp, "\n");**

**if (ground[i][j] == EOF)**

**break;**

**}**

**else**

**{**

**fprintf(fp, "%d", ground[i][j]);**

**if (ground[i][j] == EOF)**

**break;**

**}**

**}**

**}**

**fclose(fp);**

**}**

1. changeToBlock

**int changeToBrock(float now)**

**{**

**// printf("now:%f ", now);**

**float zero = 0.0;**

**int new;**

**float a, b; /\*aは商、bはあまり\*/**

**if (now > 0)**

**{**

**now -= 0.5;**

**if (now < 0)**

**{**

**/\*元々のnowが0.5より小さいのとき \*/**

**new = 0;**

２つ目の関数は、changeToBolck関数である。この関数では、x,y座標をground配列のインデックスに変換する計算を行っている。なお、インデックスは、ground[x][y]に対応している。具体的には、0<x<=0.5のときは0となり、0.5<x<=1.5のときは、1になるように計算される。

**}**

**else**

**{**

**a = (int)now;**

**b = now - a;**

**if (b > 0.0 || 0.0 > b)**

**{**

**/\*あまりがあった時\*/**

**new = a + 1;**

**}**

**else**

**{**

**new = a;**

**printf("new;%d\n", new);**

**}**

**}**

**}**

**else if (now < 0)**

**{**

**now += 0.5;**

**if (now > 0)**

**{**

**/\*元々のnowが-0.5より大きいとき\*/**

**new = 0;**

**}**

**else**

**{**

**a = (int)now;**

**b = now - a;**

**if (b > 0.0 || 0.0 > b)**

**{**

**/\*あまりがあった時\*/**

**new = a - 1;**

**}**

**else**

**{**

**new = a;**

**}**

**}**

**}**

**else**

**{**

**new = 0;**

**}**

**new += map\_max\_x;**

**return new;**

**}**

　下の図で説明する。xの値が正である場合、それぞれの値に-0.5,することで、0<x-0.5<=1.0のときは1、1.0<x-0.5<=2.0のときは２になるように、(整数部分)+1または整数部分が求めたい値になる。さらに詳しく考えると、小数点以下の値が存在する場合は(整数部分)+1をし、ぴったり整数である場合はそのまま(整数部分)=(求めたい値)であるとわかる。負の値も同様に考える。ここで、インデックスに負の値が存在しないことから、計算した値に[ground\_max\_x]を足すことで、インデックスに変換している。

グラフ

自動的に生成された説明

1. judge

**BOOL judge(char direction)**

**{**

**int viewdirection;**

**BOOL check; /\*TRUEのとき進行できる|FALSEのとき進行方向が現在位置より高いため進行できない\*/**

**int x, y, z;**

**float xbuffer = nowx;**

**float ybuffer = nowy;**

**switch (direction)**

**{**

**case 'w':**

**xbuffer += 1.0 \* cos(thetaXY \* M\_PI / 180);**

**ybuffer += 1.0 \* sin(thetaXY \* M\_PI / 180);**

**break;**

**case 's':**

**xbuffer += 1.0 \* cos((thetaXY + 180) \* M\_PI / 180);**

**ybuffer += 1.0 \* sin((thetaXY + 180) \* M\_PI / 180);**

**break;**

**case 'a':**

**xbuffer += 1.0 \* cos((thetaXY + 90) \* M\_PI / 180);**

**ybuffer += 1.0 \* sin((thetaXY - 90) \* M\_PI / 180);**

**break;**

**case 'd':**

**xbuffer += 1.0 \* cos((thetaXY - 90) \* M\_PI / 180);**

**ybuffer += 1.0 \* sin((thetaXY - 90) \* M\_PI / 180);**

**break;**

**default:**

**break;**

**}**

**printf("judge\n");**

**x = changeToBrock(xbuffer);**

**y = changeToBrock(ybuffer);**

**z = (int)nowz;**

**for (int i = x - 12; i <= x + 10; i++)**

**{**

**for (int j = y - 12; j <= y + 10; j++)**

**{**

**printf("[%d]", ground[i][j]);**

**}**

**printf("\n");**

**}**

３つ目の関数は、judge関数である。この関数では、キーボードから入力された方向に進めるか進めないかを判定する。具体的には、進行先に自分のz軸より高い座標があれば進めず、自分のz軸と同じかそれよりも低い場合は進めると判断している。原理はとてもシンプルで、進んだ先の座標をchangeToBlock関数でground配列のインデックスに変換し、そこの値を取得することで、進んだ先のz軸座標と現在地のｚ軸座標が比較できるようになる。なお、進める場合は[TRUE]を、進めない場合は[FALSE]を返す。この関数によって、当たり判定の実装が可能となった。

**if (ground[x][y] < (int)z)**

**{ /\*進行先の地面が現在位置より下に地面がある時＝着地する（nowzを下げる）\*/**

**nowz = ground[x][y];**

**setZ(nowz);**

**check = TRUE;**

**}**

**else if (ground[x][y] == (int)z)**

**{ /\*進行先の地面が現在位置と同じ時＝進行する\*/**

**nowz = ground[x][y];**

**setZ(nowz);**

**check = TRUE;**

**}**

**else**

**{**

**check = FALSE;**

**}**

**printf("key:%c, もし進めたらground[%d][%d]\n", direction, x, y);**

**return check;**

**}**

1. setZ

**void setZ(float ground)**

**{**

**nowz = ground;**

**float bufferZ = lookfromz;**

**lookfromz = nowz + 1.75;**

**bufferZ = lookfromz - bufferZ;**

**lookatz += bufferZ;**

**}**

　最後の関数は、setZ関数である。この関数は[nowz]をもとに、[lookfromz/lookatz]に自動で代入するものである。[lookfrom/lookatz]の値と[nowz]の値が、[↑/↓]キーを押している場合を除き、常に同じ差があることを利用している。