情報メディア実験A

物理エンジンを使ったアプリケーション開発

筑波大学情報学群
情報メディア創成学類
藤澤誠

実験の目的

物理シミュレーションエンジンを利用したアプリケーションが作れるようになる

- 教員による説明回+講義ページの説明を理解して練習問題を解く(実装する)
- 物理エンジンを用いた簡単なアプリケーションを作成

実験スケジュール

- 毎週 水3,4限＆金5,6限
- テーマ内スケジュール
  1. ガイダンス＆事前知識(C++)説明：4/16
  2. 物理シミュレーションとは?：4/21, 23, 28
  3. 物理エンジンとは？ ：4/30, 5/7, 12
  4. 剛体間の衝突判定,衝突応答：5/14, 21, 26, 28
  5. 剛体間リンク：6/2, 4, 9, 11
  6. 3Dモデル読み込みと弾性体：6/16, 18, 23, 25
  7. アプリケーション開発：6/30, 7/2, 7, 9, 14, 16, 21
  8. 成果発表会：7/28

赤太字は説明回, 7/22は発表準備＆レポート作成回
5/7(金)は水曜授業日, 5/19(水)は春A期末試験日

教室とWebページ

- 教室
  ⇒ COVID-19対策ですべてTeamsによるリモート講義
  - 4/16,21,30,6/2,16,30：パワーポイントを使った説明
  - 7/28：各自のアプリケーション作成成果をTeamsのミーティングで画面共有機能を使ってプレゼン
  - 上記以外：Teamsミーティングを開くので各自開発＆質問時間
- Webページ
  http://slis.tsukuba.ac.jp/~fujis/lecture/iml/
  講義資料はTeamsも使って共有します.

環境構築

実験説明ページ及びサンプルコードは**Windows+Visual Studio 2017以降**を想定している．ただし，Mac環境でもXcodeなどで動くことは確認している.

- Windows環境
  無料版のVisual Studio Communityをインストールしてもらえれば説明ページに書かれていることは問題なく実行可能です(VS2019Communityで動作確認済み)
  → 詳しいインストール方法はTeamsの「ファイル」にある「**Visual Studio 2019 Communityインストール方法.pdf**」参照
- Mac環境
  本実験テーマで使う物理エンジンbulletのMacでのインストール(ビルド)については以下のページなどが参考になる.
  http://blog.livedoor.jp/tek_nishi/archives/10156724.html
  - 最初のサンプルコード(物理エンジンを使っていないもの)についてはOpenGL(GLUT)が必要となる．そちらについては以下のページなどが参考になる.
    https://qiita.com/totepo18/items/b2a6bdb8054f8c9a1cbe

大学のリモートデスクトップの使い方1

1. 全学計算機システムのページにブラウザでアクセス
   https://www.u.tsukuba.ac.jp/
2. 左バーから「ログイン・パスワード」→「リモートアクセス」をクリック
3. リモートアクセスのページに移動するので,
   「全学計算機システムのWindowsデスクトップを利用する」から自身の環境にあったリンクをクリック(Windowsなら4, Macなら5)

大学のリモートデスクトップの使い方2

4. それぞれの環境に合わせたリモートデスクトップへのアクセス方法が書かれているので，手順に従ってリモートアクセスする．

2-4. Windows の「リモートデスクトップ接続」で接続

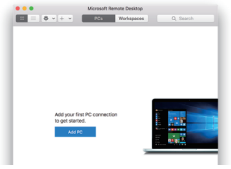
[接続]

1. デスクトップ左下の「スタート」ボタン (Windows の口ゴが描いてあるボタン) をクリックし，[Windows アクセサリ]-[リモートデスクトップ接続] をクリックします．

2-5. Mac OS 上の「Microsoft Remote Desktop」で接続

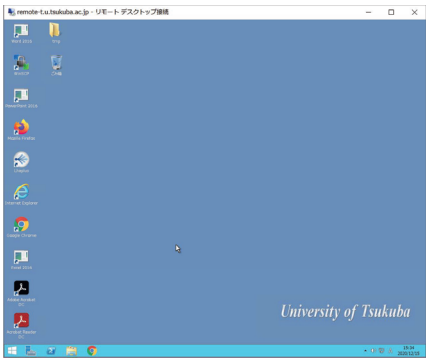
[接続]

1. 最新の Microsoft Remote Desktop を Mac App Store からダウンロード・インストールします．iTunes からのインストールのため、Apple ID が必要です。
※以下の説明は、Microsoft Remote Desktop Version 10.3 を使用しています。
2. Microsoft Remote Desktop を起動すると、下図のようなウインドウが表示されるので「Add PC」をクリックします．



大学のリモートデスクトップの使い方3

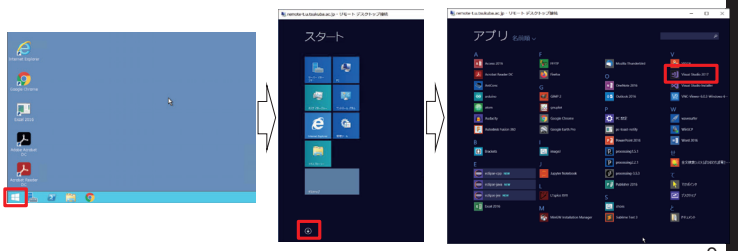
5. 印刷枚数などに関する注意事項が出たらOKをクリックし，下のような画面が出ればOK(デスクトップのアイコンは異なっていると思います)．



リモートデスクトップを用いた実験1

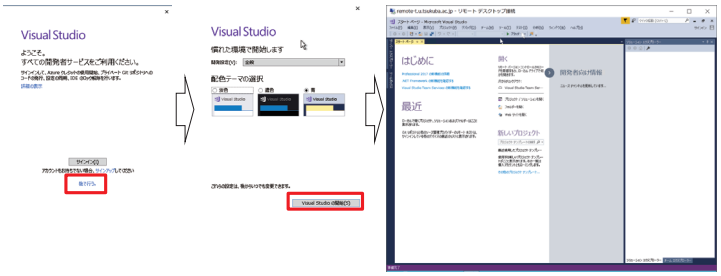
リモートデスクトップで全学計算機のWindows環境にアクセスできたら，
その環境で**Visual Studio 2017(以下VS2017)**を使って実験を進めていきます．
以下はVS2017での演習の進め方の説明です．

1. 左下のウィンドウアイコンをクリックしてスタートメニューを出す．
2. 左下に下矢印アイコン をクリックしてアプリ一覧を出す．
3. アプリ一覧から「Visual Studio 2017」をクリックする．



リモートデスクトップを用いた実験2

4. Visual Studioの初期設定画面が出たら「後で行う」→「Visual Studioの開始」をそれぞれ選択(アカウントがある人はサインインしてもOK. 実験を行う上ではなくてもOK)
5. 右下のようなVisual Studioのウィンドウが出たらOK．

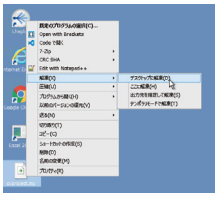


リモートデスクトップを用いた実験3

6. サンプルプロジェクトファイル(iml_physics.zip)を以下のページの「1. 物理シミュレーションとは？」からリモートデスクトップにダウンロードする．
サンプル配布ページ：http://slis.tsukuba.ac.jp/~fujis/lecture/iml/#schedule

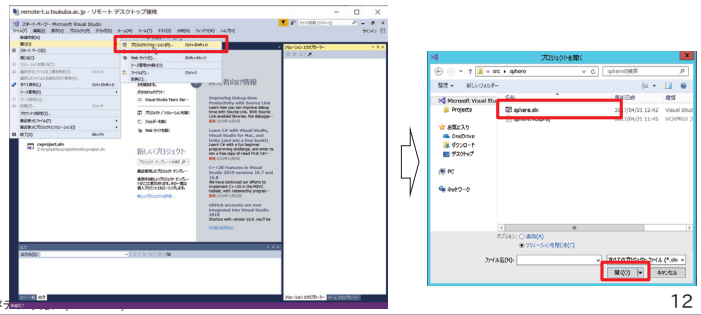
リモートデスクトップ内でmanabaにアクセスしてダウンロードするか，
自身の環境でダウンロードしたものを転送する．
それぞれの環境の転送方法は https://www.u.tsukuba.ac.jp/remote/ 参照

7. ダウンロードしたファイルを右クリックして，「解凍」でファイルを解凍する
(解凍場所は任意)



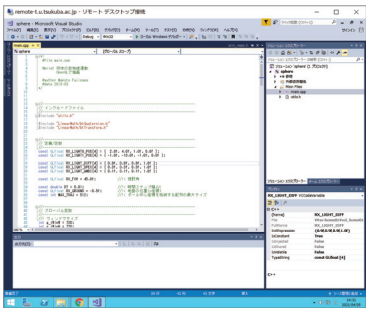
リモートデスクトップを用いた実験4

8. Visual Studio 2017を開き，「ファイル」→「開く」→「プロジェクト/ソリューション」をクリックする．
9. 「プロジェクトを開く」ウィンドウが出るので，手順7で解凍したフォルダ内の iml_physics/src/sphere/sphere.slnファイルを選択して，「開く」をクリックする．
(セキュリティ警告が出たらそのままOKをクリック)．



リモートデスクトップを用いた実験5

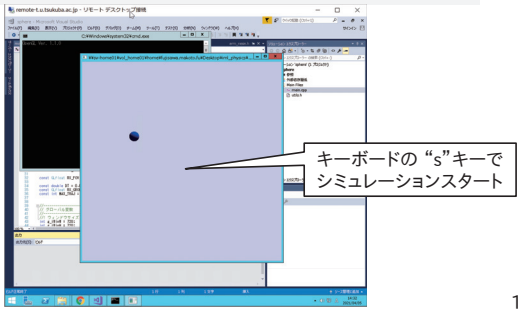
10. VS2017ウィンドウの右にある「ソリューションエクスプローラー」から，sphereプロジェクトをダブルクリックして開き，Main Filesの中のmain.cppファイルを開く．
11. コードを確認したら，「ビルド」メニュー→「sphereのビルド」を選択，下の出力領域にビルドの様子が出るので，「すべてビルド: 1 正常終了…」と出ればOK．



リモートデスクトップを用いた実験6

12. ビルドが正常に終了したら，「デバッグ」メニュー→「デバッグなしで開始」をクリックする(「デバッガを使う場合は「デバッグの開始」でもOK)．
13. 下図のように青いボールが描画されたウィンドウが表示されればOK．

実験ページ(http://slis.tsukuba.ac.jp/~fujis/lecture/iml/)にアクセスして，
実験を進めていこう！



C++の基本

- 本実習で使う物理エンジン(bullet physics)はC++で書かれたライブラリ(Pythonラッパあり)
  - C言語の拡張としてのC++の基本
  - メモリの確保と解放
  - オブジェクト指向の基本

ライブラリを使うための最低限の機能だけを教えます．

Hello World!(C言語版)

List.1

```c
#include <stdio.h>
int main(void)
{
  /* Hello の画面出力World */
  printf("Hello␣World!\n");
  return 0;
}
```


Hello World!(C++版)

List.2

```c
#include <iostream>
//名前空間の設定
using namespace std;
int main(void)
{
  // Hello の画面出力World
  cout << "Hello␣World!" << endl;
  return 0;
}
```


標準入出力

- iostream

標準出力 : cout	⇔	printf
標準入力 : cin		scanf
標準エラー出力 : cerr		fprintf(stderr, ···)

- 使い方

```c
float a;
int b;
cin >> a;
cin >> b;
cout << "a␣=␣" << a << endl;
cout << "b␣=␣" << b << "␣=␣0x" << hex << b << endl;
```

>>, <<の本来の使い方はシフト演算
⇒演算子のオーバーロード

cinのエラー処理

- エラー処理

  cinは内部にエラーフラグ変数を持つ

```
int x;
cin >> x;
while(!cin){
  cin.clear();
  cin.ignore(INT_MAX, '\n');
  cout << "再入力 : ";
  cin >> x;
}
```

エラーフラグのチェック

エラーフラグのリセット

入力バッファのクリア

再入力

ファイル入出力 1

- fstream (ifstream, ofstream)

ファイル出力 : ofstream	fprintf
ファイル入力 : ifstream	fscanf
ファイル入出力 : fstream	FILE

- ファイル入出力の手順

ファイルを開く

開けたら

データをファイルに入力 or 出力

ファイルを閉じる

ファイル入出力2

```
ofstream fo;
fo.open("output.txt");
if(fo){
  fo << "データの出力" << endl;
  fo.close();
}
else{
  //エラー処理
}
```


変数の宣言

List.3

```
#include <iostream>
using namespace std;
int main(void)
{
  //変数の宣言
  for(int i = 0; i < 10; ++i){
    double a = (double)i*i;
    cout << a << endl;
  }
  return 0;
}
```


関数のデフォルト引数

List.4

```
#include <iostream>
using namespace std;
//デフォルト引数
void def_test(int d = 0)
{
  cout << "d = " << d << endl;
}
int main(void)
{
  def_test();
  def_test(1);
  return 0;
}
```


関数のオーバロード

List.5

```
#include <iostream>
using namespace std;
//関数オーバロード
int func(void)
{
  int d;
  cout << "Input Integer Number : ";
  cin >> d;
  return d;
}
void func(int d)
{
  cout << d << endl;
}
int main(void)
{
  func(func());
  return 0;
}
```

引数の参照渡し

List.6

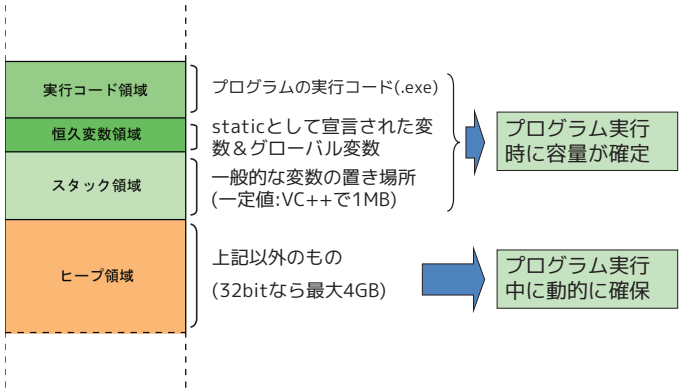
```
void swap(int *a, int *b)
{
  int c;
  c = *a;
  *a = *b;
  *b = c;
}
```

List.7

```
void swap(int &a, int &b)
{
  int c;
  c = a;
  a = b;
  b = c;
}
```


メモリについて

- メモリマップ



スタック領域

- スタック領域は何のためにあるのか？

  レジスタの数を超えた変数が使用されたときに用いられる

  - スタックの利点：割り当てや開放の手間がないので高速
  - スタックの欠点：スタックのサイズはあまり大きくできない

  ⇩

  - ヒープの利点：サイズ制限なし(ただしメモリ容量による)
  - ヒープの欠点：OSによるメモリ管理コードのため低速

メモリの動的確保

- malloc,free (C言語)

```
void *malloc(size_t size)
void *calloc(size_t n, size_t size)

int *p;
p = (int*)malloc(10*sizeof(int));
--------pを使った処理 ---------
free(p);
```

- new,delete (C++)

```
int *p;
p = new int[10];
--------pを使った処理 ---------
delete [] p;
```

確保した領域は必ず開放する

(STLのvectorやbulletのbt*Arrayの場合は自動開放)

番外:STLによる動的配列

- STL(標準テンプレートライブラリ)による動的配列

```
#include <vector>
using namespace std;
void main(void){
    vector<int> x;
    x.resize(5, 0); // 配列サイズ変更(size:5)して，値0で初期化
    x.push_back(10); // 配列に値を追加(size:6)

    // 反復
    for(int i = 0; i != x.size(); ++i) cout << x[i] << endl;
    // イテレータを使った反復
    for(vector<int>::iterator i = x.begin(); i != x.end(); ++i)
        cout << *i << endl;
}
```


オブジェクト指向とは

- オブジェクト指向

「関連するデータの集合と，それに対する手続き(メソッド)を「オブジェクト」と呼ばれる一つのまとまりとして管理し，その組み合わせによってソフトウェアを構築する手法」

(IT用語辞典e-Wordsより)

- カプセル化(Encapsulation)
- 継承(Inheritance)
- 多態性(Polymorphism)

構造体の復習

List.8

```
#include <iostream>
using namespace std;
typedef struct Vec3
{
  float x, y, z;
} Vec3;
int main(void)
{
  Vec3 v;
  v.x = 1;
  v.y = 2;
  v.z = 3;

  return 0;
}
```


はじめてのクラス

List.9

```
#include <iostream>
using namespace std;
class Vec3
{
  float x, y, z;
};
int main(void)
{
  Vec3 v;
  v.x = 1;
  v.y = 2;
  v.z = 3;

  return 0;
}
```

メンバ変数

オブジェクト

アクセスコントロール 1

- private,public,protected

  List.9のコードはコンパイルエラーとなる

  ↓ なぜか？

  アクセスコントロール
  - publicメンバ：外部からアクセス可能
  - protectedメンバ：継承先からアクセス可能
  - privateメンバ：クラス内からのみアクセス可能

  デフォルトのアクセスコントロール
  - 構造体：public
  - クラス：private

アクセスコントロール2

List.9

```
#include <iostream>
using namespace std;
class Vec3
{
public:
  float x, y, z;
};
int main(void)
{
  Vec3 v;
  v.x = 1;
  v.y = 2;
  v.z = 3;

  return 0;
}
```


メンバ関数

- 関数をクラスのメンバに

```
int main(void)
{
  Vec3 v;
  v.input(1.0f, 2.0f, 3.0f);
  v.output();
  return 0;
}
```

List.10

メンバ関数

```
class Vec3
{
public:
  float x, y, z;
  void input(float x0, float y0, float z0);
  void output(void);
};
void Vec3::input(float x0, float y0, float z0)
{
  x = x0; y = y0; z = z0;
}
void Vec3::output(void)
{
  cout << "v = (" << x << ", ";
  cout << y << ", ";
  cout << z << ")" << endl;
}
```


カプセル化

- カプセル化

  アクセスコントロールでクラス内部の詳細をブラックボックス化

  メンバ変数はpublicにしない

```
class Vec3
{
private:
  float x, y, z;
public:
  void input(float x0, float y0, float z0);
  void output(void) const;
};
```

コンストラクタとデストラクタ

- コンストラクタ(Constructor)

  オブジェクトを生成したときに呼ばれ，主に初期化を行う

  ```
  public:
    クラス名 (引数)
  ```

  返値なし，引数あり
  オーバーロードにより複数存在可

- デストラクタ(Destructor)

  オブジェクトを破棄したときに呼ばれ，主に後処理を行う

  ```
  public:
    ~クラス名 ()
  ```

  返値なし，引数なし
  複数存在不可

コンストラクタとデストラクタ

List.12

```
class Vec3
{
private:
  float x, y, z;
public:
  //コンストラクタとデストラクタ
  Vec3()
  {
    x = 0.0f; y = 0.0f; z = 0.0f;
  }
  Vec3(float x0, float y0, float z0){
  {
    x = x0; y = y0; z = z0;
  }
  ~Vec3(){}
};
```

デフォルトコンストラクタ

インスタンス

List.13

```
int main(void)
{
  Vec3 *vp;
  vp = new Vec3(1.0f, 2.0f, 3.0f);
  vp->output();
  delete vp;
  return 0;
}
```


まとめ

- C++で拡張された機能

  オーバーロード，参照渡し，newとdelete

- オブジェクト指向：クラス

  アクセスコントロール，コンストラクタとデストラクタ，カプセル化

- カプセル化(Encapsulation)
- 継承(Inheritance)
- 多態性(Polymorphism)

興味のある人はWebページにある資料「オブジェクト指向言語 C++ 後編」を読んでみてください．