Panda3D 基礎技術

3Dゲームエンジンの基本概念

Panda3D

- オープンソースの3Dゲームエンジン
- C++で実装され、Pythonバインディングを提供
- Disneyが開発し、2002年にオープンソース化
- Carnegie Mellon大学が共同で開発
- クロスプラットフォーム対応（Windows、Linux、macOS）

Panda3D の主要な機能

- **既定のマウス操作**　　右ボタン，左ボタンを押しながらマウスを動かすと，シーン全体が動く
- **3次元モデルファイルの読み込み**　**loadModel**
- **3次元モデルの配置**
  - 位置　　**setPos**
  - 拡大縮小　　**setScale**
  - 回転　　**setQuat**
- **イベントハンドラの登録**　　**accept**
- **キーコード**の例　“A”, “space”, “enter”, “arrow_left”, “arrow_up”, “arrow_down”, “arrow_right”　など
- **オブジェクトの位置取得と操作**　　**set, get**
- 自動で動かす　　**taskMgr.add**

ゲームループとフレーム

ゲームループの概念

- 3Dゲームエンジンの繰り返し処理
- ゲーム実行中、毎秒数十～数百回実行
- 1回の繰り返し = 1フレーム

FPS（Frames Per Second）

- フレームレートの指標
- 一般的なゲームでは 60fps が目安

デルタ時間（Delta Time）

- 前フレームから現在フレームまでの経過時間（秒単位）
- フレームレート非依存の動作を実現

例：1秒間に10度回転させる場合 → 毎フレーム「10 × dt」度ずつ回転

Panda3Dプログラムの基本構造

1. モジュールのインポート

↓

2. ShowBaseクラスを継承したアプリケーションクラスの定義

↓

3. オブジェクトの作成と設定

↓

4. app.run()でゲームループ実行

継承とは

- 既存のクラス（ShowBase）の機能を引き継いで新しいクラスを作成する仕組み
- **class MyApp(ShowBase):** と記述することで、ShowBaseの機能を持つMyAppクラスを定義

ShowBaseクラスの継承

- Panda3Dアプリケーションの基盤
- レンダリング、入力管理、タスク管理を担当

コード例

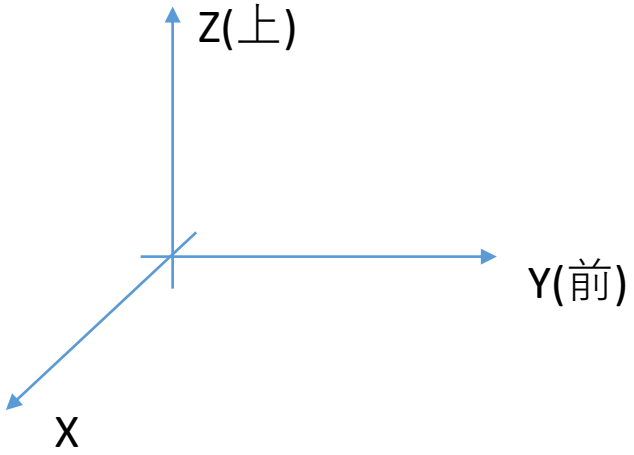
```
from direct.showbase.ShowBase import ShowBase
class MyApp(ShowBase):
    def __init__(self):
        ShowBase.__init__(self)
        # ここにオブジェクトの作成・設定を記述
app = MyApp()
app.run()
```

3D直交座標系の基礎

3次元直交座標系

- XYZ軸による位置表現
- 3つの数値(x, y, z)で空間内の点を特定

Panda3Dの座標系（Z-up右手座標系）



ベクトルと座標系の種類

- **ベクトル**
  - 大きさと方向を持つ量
  - (x, y, z)の3成分で表現
  - 位置、速度、加速度などを表現
- **座標系の種類**

1. **絶対座標（ワールド座標）**
   - 原点(0, 0, 0)を基準とした座標
2. **相対座標（ローカル座標）**
   - 親オブジェクトを基準とした座標
   - 親が移動すると子も一緒に移動

- **シーングラフ**
  - オブジェクトを階層構造（木構造）で管理
  - self.renderがルートノード
  - reparentTo()で親子関係を設定

3つの基本変換

- **移動（Translation）**

  cube.setPos(0, 5, 1) ＃X=0, Y=5, Z=1に
- **回転（Rotation）**
  - Heading（H）：Y軸周りの回転
  - Pitch（P）：X軸周りの回転
  - Roll（R）：Z軸周りの回転
  - 角度は 0～360度で指定

  cube.setH(45) ＃Y軸周りに45度回転
- **スケール（Scale）**
  - オブジェクトの大きさを変更
  - 1.0で元のサイズ、2.0で2倍、0.5で半分

  cube.setScale(2, 1, 1) ＃X方向に2倍```

カメラと視点制御

- **視点（Camera Position）**
  - 観察者の位置
  - camera.setPos(x, y, z) で設定
- **注視点（Look-at Point）**
  - カメラが向いている目標点
  - camera.lookAt(object) で設定

カメラ位置

●　--------　視線方向　　○　注視点

メッシュ

- **メッシュ（Mesh）**
  - 3Dオブジェクトの形状を定義する頂点と面の集合
  - loader.loadModel() で読み込み
- **Panda3Dの基本図形**
  - models/box：立方体
  - models/plane：平面

色の表現 (RGB)

- Red、Green、Blueの3成分
- 各成分は0.0～1.0の範囲
- setColor(R, G, B, A) で設定
- 例：(1, 0, 0)は赤、(0, 1, 0)は緑、(1, 1, 1)は白色

```
cube.setColor(1, 0.5, 0, 1)
```

ここまでのまとめ

- ゲームループ
- デルタ時間によるフレームレート非依存
- 3D空間の表現: ３次元直交座標系
- シーングラフ：オブジェクトの階層管理
- オブジェクトの制御：移動、回転、スケール
- メッシュ
- 色

ゲームエンジンの利用により３次元アプリの開発が容易になる

ライティングとシェーディング

環境光（AmbientLight）

- 全方向から均一に照らす光
- 影を作らず、シーン全体を明るく照らす

```
ambient = AmbientLight('ambient’)
ambient.setColor((0.4, 0.4, 0.4, 1))
ambient_np = self.render.attachNewNode(ambient)
self.render.setLight(ambient_np)
```

指向性光源（DirectionalLight）

- 太陽光のように特定の方向から平行に照らす光
- 明確な影を作り、立体感を表現

```
sun = DirectionalLight('sun’)
sun.setColor((0.8, 0.8, 0.8, 1))
sun_np = self.render.attachNewNode(sun)
sun_np.setHpr(45, -60, 0)
self.render.setLight(sun_np)
```

エンティティの生成と制御

3D空間内のオブジェクトをツリー構造で管理

例)

vehicle (親ノード)

変換の伝播

- 親ノードの移動 → 全ての子ノードも移動
- 子ノードの移動 → 親ノードは影響を受けない

入力処理

2つの入力方式

- **イベント駆動**：単発の入力、ジャンプ、攻撃など

```
def jump(self):
    if not self.is_jumping:
    self.jump_force = 5
    self.is_jumping = True
self.accept('space', self.jump)
```

- **キー状態管理**：継続的な入力、移動、回転など

```
self.keys = {'w': False, 'a': False, 's': False, 'd': False}
def setKey(self, key, value):
    self.keys[key] = value
self.accept('w', self.setKey, ['w', True])
self.accept('w-up', self.setKey, ['w', False])
# 状態チェック
If self.keys[‘w’]:
    self.player.setY(self.player.getY() + self.speed * dt)
```

物理演算

- ベクトル：位置、速度、加速度を表現
- 重力加速度：地表では約-9.8 m/s²（下向き）

物理シミュレーションの2段階計算

```
self.gravity = -9.8
# ステップ1：加速度を速度に加算
velocity.z += self.gravity * dt
# ステップ2：速度を位置に加算
new_pos = box.getPos() + velocity * dt
box.setPos(new_pos)
```

衝突判定

オブジェクトが接触しているかを座標で判定。

```
collision_z = 0 # 地面の高さ
if box.getZ() <= collision_z:
    box.setZ(collision_z) # 位置を補正
    velocity.z = -velocity.z * 0.5 # 速度を反転（反発）
    # 停止判定
    if abs(velocity.z) < 0.1:
        velocity.z = 0
```

ここまでのまとめ

ライティングとシェーディング

- 環境光と指向性光源で立体感を演出

エンティティの生成と制御

- シーングラフの階層構造
- 親子関係による変換の伝播

入力処理

- イベント駆動とキー状態管理の使い分け

物理演算

- ベクトル、速度、加速度- 運動方程式による位置更新
- 衝突判定

ゲームエンジン応用の4つの重要ポイント

フレームレート非依存設計

↓ dt（経過時間）を提供

数値計算の離散化

↓ 計算結果を生成

動的データ構造の管理

↓ オブジェクトの追加・削除

動的ジオメトリの更新

↓ 視覚的表現

インタラクティブで物理的に正確なリアルタイムアプリケーションを構築

フレームレート非依存の設計

- ゲームやシミュレーションは様々な環境で実行される。**フレームレートが変動**しても、動作が正しく時間に沿ったものである必要がある。

```
def update(self, task):
    current_time = globalClock.getFrameTime()
    dt = current_time - self.prev_time  # 前フレームからの経過時間
    self.prev_time = current_time

    # 移動量 = 速度 × 時間
    self.player.setY(self.player.getY() + self.speed * dt)
```

動的データ構造の管理

- ゲームやシミュレーションでは、オブジェクトの追加・削除が発生する。

```
# 誤り：イテレーション中の元リスト変更
for item in self.collectibles:
    if condition:
        self.collectibles.remove(item)  # 危険
# 正解：リストのコピーをイテレート
for item in self.collectibles[:]:  # [:] でコピー作成
    if condition:
        item.removeNode()  # シーングラフから削除
        self.collectibles.remove(item)  # リストから削除
```

シーングラフ（描画対象）とデータ構造（管理リスト）の両方を同期して更新

数値計算の離散化

- 連続的な物理現象をコンピュータで扱うには、時間と空間を離散的な点で近似する必要がある。
- 連続的な微分方程式を差分方程式に変換

連続：速度 = d(位置)/dt

離散：速度 ≈ (現在位置 - 前回位置) / dt

velocity = (self.current[i][j] - self.previous[i][j]) / dt

連続：ラプラシアン = ∂²u/∂x² + ∂²u/∂y²

離散：上下左右4点の加重平均

laplacian = (u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - 4*u[i][j]) / (spacing²)

動的ジオメトリの更新

- 静的なモデル読み込みだけでは表現できない動的な形状変化（波、変形、破壊など）を実現。そのために、頂点データを直接操作

```
# 1. 動的更新可能なメッシュ作成（初期化時）
format = GeomVertexFormat.getV3n3c4()
vdata = GeomVertexData('mesh', format, Geom.UHDynamic)  # UHDynamic指定

# 2. 頂点データの取得（更新時）
geom = self.mesh_node.modifyGeom(0)
vdata = geom.modifyVertexData()

# 3. 頂点位置の書き換え
vertex = GeomVertexWriter(vdata, 'vertex')
for i in range(grid_size):
    for j in range(grid_size):
        z = self.current[i][j]  # 計算結果
        vertex.setData3(x, y, z)  # 頂点位置を更新
```