シューティングゲーム　『アインシュタインインベーダー』の作成

大阪工業大学　情報科学部　情報システム学科
B15049 関谷光一郎

2019 年 2 月 11 日

目 次

1　序論

1.1　背景

万有引力の法則とは、２つの物体の間には，２つの物体の質量に比例し，２つの物体の距離の 2 乗に反比例する引力が，作用するという法則である．

これはケプラーの法則を説明するために，ニュートンが仮定したものである．運動方程式に万有引力（中心力）を適用することで，ケプラーの法則が導出される．後にアインシュタインの一般相対性理論では，重力の正体は質量を持つ物体が引き起こす時空の歪みであると説明された．

1.2　本研究の目的

本研究の目的は，実際の物理法則に基づいた，惑星の重力を受けるシューティングゲームをゲーム作成ツールである Unity を使って実現することである．

具体的なゲーム内容は，プレイヤーが的を狙って球体を発射するというシンプルなシューティングゲームに，万有引力の働く惑星を追加し，球体の軌道を邪魔する障害物となる．よって，プレイヤーは重力で球体の軌道が変わることも考慮して狙う必要がある．さらに障害物としてブラックホールを実装し，ブラックホールの位置を確かめるためのレーザーを発射できるようにする．プレイヤーはこのレーザーの軌道を観察し，ブラックホールの位置を推定することができる．　教育的な観点から，別の視点から球体の動きを観察できるようなカメラと，動く球体からみたカメラを追加する． また，球体がどのような運動をしているかを分かりやすくするため、球体の通った後が線で描写されるようにする．

本研究を達成することにより，2 つの利点がある．1 つ目は，万有引力のリアルなシミュレータを作成でき，様々な条件での実験が行えるということ．2 つ目は，シューティングゲームとして遊ぶ熟達者にとっても，万有引力を一歩進めた，相対性理論の影響を考察する機会を与えることである．レーザーの軌道でブラックホールの位置を推定できる．　この 2 つの利点があることにより，このゲームの対象者は子供から大人まで幅広く，様々な目的で利用することができる．

1.3　本研究の構成

本論文の構成は次のようになっている．

第 2 章では，Unity で作成するゲーム内容について述べる．

第 3 章では，ゲームに実装する万有引力の運動方程式について説明し，その動き方について考察する．

第 4 章では，ブラックホールと，ブラックホールの位置を推定するレーザーの運動方程式について説明し，その軌道について考察する．

最後に，第 5 章で本論文で行ったまとめを述べる．

2　Unity でのゲーム作成

2.1　Unity について

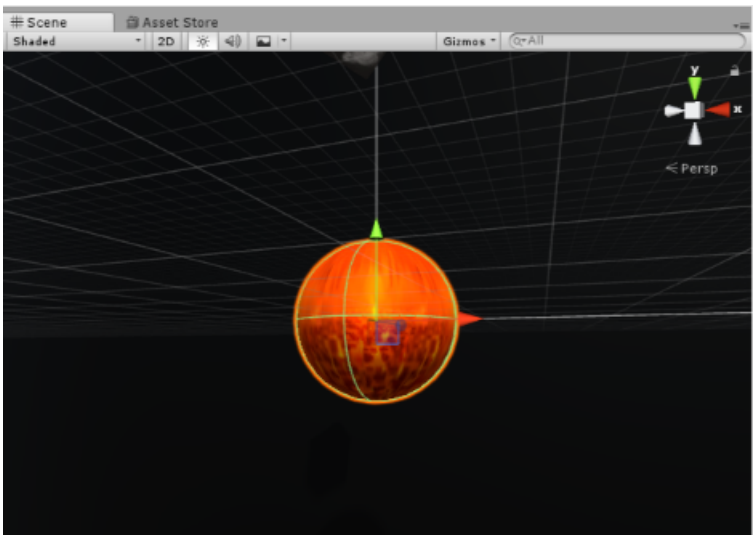

図 1: Unity 画面

Unity とはユニティ・テクノロジーズ社が提供する、２ D，３ D 両方に対応したゲーム開発プラットフォームである. PC やスマートフォンなど様々なプラットフォームに出力することが可能で，ゲーム開発に必要な様々な機能が備わっている. 例えばタブの GameObject　→　3D Object　→　Sphere　で球体などのオブジェクトが図 1 のように簡単に作成できる．OS は windows，Mac に対応しており，年商１０万ドル以下の場合は無料で利用できる.（https://store.unity.com/ja/download?ref=personal　からダウンロード）現在スクリプトの言語は C ＃のみの対応であるため，今回は C ＃でプログラムを作成する．

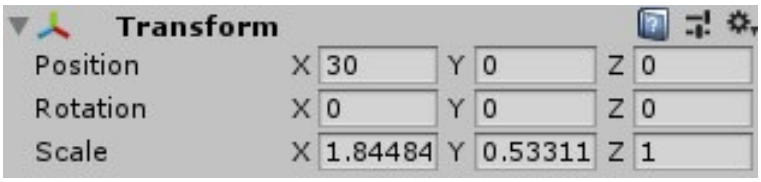

図 2: Transform 設定画面 I

ゲームオブジェクトを作成すると図 2 のような Transform の GUI が追加される．Position，Rotation，Scale で，位置，回転角度，大きさを設定することができる．

2.2　ゲーム概要

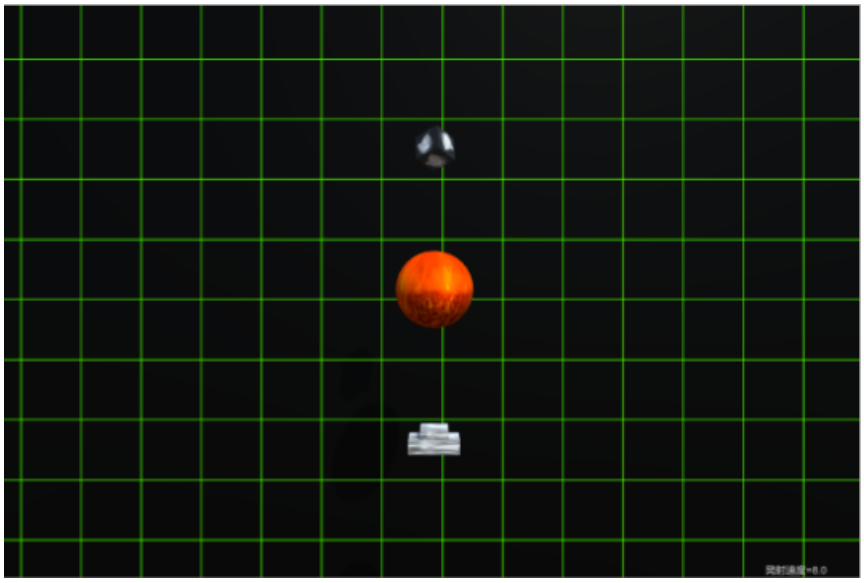

図 3: Unity 画面

まず作成したほぼすべてのオブジェクトに rigidbody を適応させる.rigidbody とは，物理演算を行うために必要な機能で，物体の質量，空気抵抗などを設定できる. 適応の仕方は，Add Component → rigidbody　を選択する.

図 3 の下方にある物体が発射台で，これを左右に移動させ，任意のタイミングで球を発射させ，上部にある正方形の的に当てるゲームである．プレイヤーは中央の球体の万有引力を考慮して狙う必要がある.

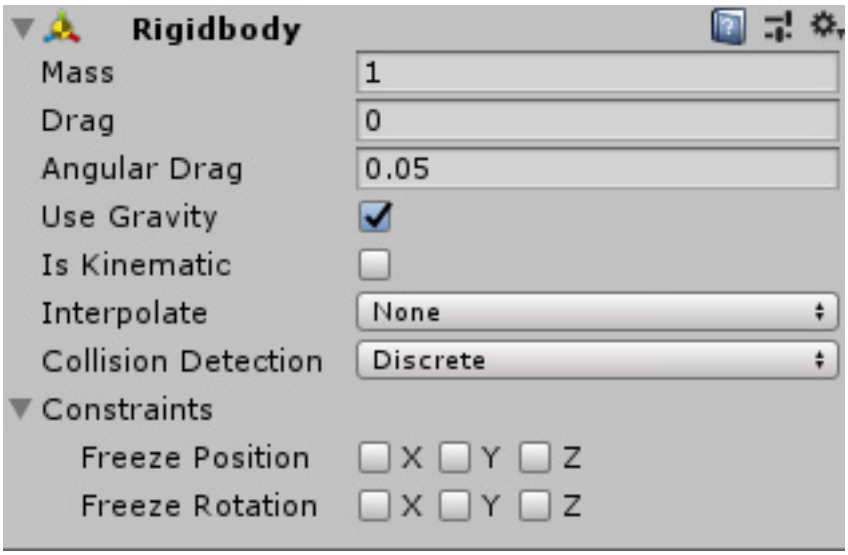

図 4: rigidbody 設定画面

rigidbody を適応させると，図 4 のような rigidbody の GUI が追加される．Mass は kg 単位の質量，Drag は空気抵抗，Angular Drag はオブジェクトが回転する際の空気抵抗，Use Gravity を有効にすると，オブジェクトに $-y$ 方向の重力の影響を与える．今回はスクリプトでオブジェクトを重力源とするため無効にする．Is Kinematic を有効にすると，あらゆる rigidbody の影響を受けなくなる．Interpolate は rigidbody の動きが不安定なとき，オ

プションを一つ選択して改善させる．例えばオブジェクトが高速で移動して，別のオブジェクトをすり抜けてしまう不具合が発生した場合，Collision Detection を選択すると改善される．ただし処理が重くなる原因となるため，改善が不要な場合は基本的に None でよい．Constraints は rigidbody の動きに関する制限．Freeze Position は選択した座標の rigidbody による移動を停止する．今回は Z 方向への移動は不要なため，Z にチェックを付ける．Freeze Rotation は選択した座標の rigidbody による回転を停止する．

2.2.1　発射台の作成

図 3 下部にある発射台は Cube オブジェクトにスクリプトを適応させてプレイヤーが操作できるようにする.Add Component の New script を選択することでスクリプトを実装できる.

transform.right * Input.GetAxisRaw(”Horizontal”) * speed で発射台を任意の方向に動かすことができる. transform.right は右方向に，Input.GetAxisRaw(”Horizontal”) は矢印キーの右を押せば 1，左を押せば-1，何も押さなければ 0 を取得する. つまり左を押せば，右方向の transform.right は左方向となる.speed は float で宣言し，値を代入する. 発射台の動く速さがこの数値で決まる.

2.2.2　発射する球体の作成

次に，球を発射するスクリプトを作成する. 発射する球のオブジェクト Sphere を作成し，Kyu とする.Rigitbody を適応させ，プレハブを作成する. プレハブとは，作成済みのオブジェクトを複製する機能であり，同じ性質のオブジェクトを多数作成することができるようになる.Project ビューの　Create　→　Prefab　を選択すると，Project ビューに New Prefab が作成されている. そこに上記で作成した Kyu をドラッグ＆ドロップすることで Kyu のプレハブが作成できる.

次に作成したプレハブを発射させるスクリプトを作成する. 条件分岐 Input.GetKeyUp(KeyCode.Z) により,「Z」キーが押して放したときの発射台の位置に球体を出現させる．hassya を public Transform で宣言し，
kyu.transform.position = hassya.position で球の出る位置を発射台に設定する．
GameObject kyus = GameObject.Instantiate(kyu) as GameObject でプレハブの Kyu を複製した Kyus を発射する．

また，発射速度を変更できようにする．public float speed を宣言し，条件分岐 Input.GetKeyDown(KeyCode.Z) で「Z」キーを押したとき，speed に 0.0f を代入する．次に「Z」キーを押している間の条件分岐 Input.GetKey(KeyCode.Z) で speed を 0.2 ずつ加算させる．また条件分岐に speed <= 14.9f を加え，speed に上限を設けた．

2.2.3　球体の速さの表示

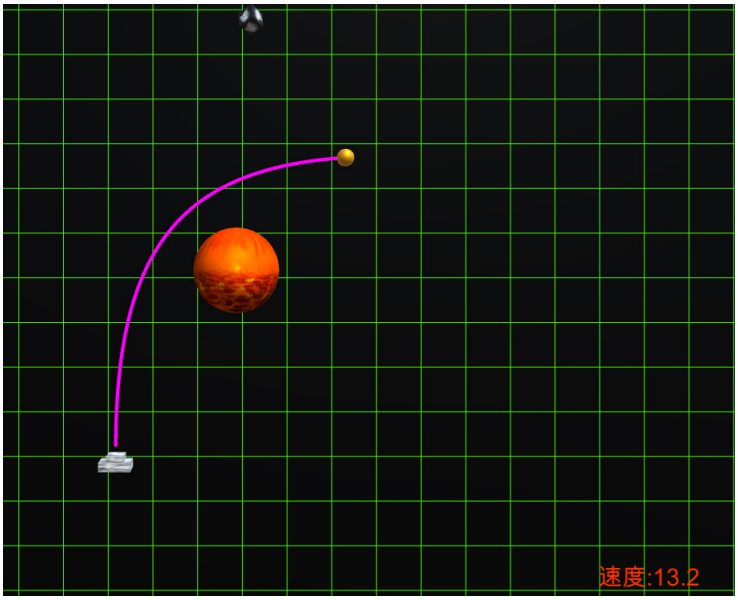

図 5: 速度表示

図 5 は，適当に球体を発射してみたときの図である．右下に赤文字で速度が表示されてることがわかる．作成方法は，GameObject → UI → Text で自動的に Canvas とその子オブジェクトとして Text が生成される．Canvas は UI が配置，描画される抽象的な領域である．Text に文字が画面上に表示される．変数の文字を表示したい場合はスクリプトを作成し，public GameObject Sobject を宣言し，null を代入しておく，そして 2.2.2 章の発射速度の変更処理をもう一度記述し，最後に
Text speedtext = Sobject.GetComponent<Text>();
speedtext.text = ”速度:” + speed.ToString(”N1”); で speed の値を表示する．

2.2.4　的の作成

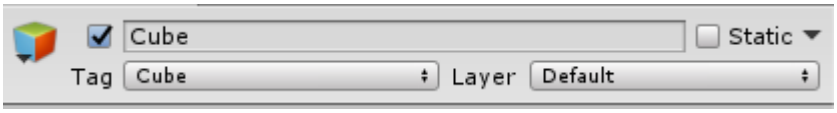

図 6: タグ

GameObject から的となる Cube を作成し，void OnCollisionEnter(Collision collision) 関数を使い，球体が的に触れたときの，衝突判定を設定する．衝突した物が球体のときに消える処理を行うため，Kyu と Cube のタグを図 7 から Cube に変更しておき，条件分岐 collision.gameObject.tag == ”Cube”のとき，Destroy(this.gameObject) で的が削除される．

2.3　画面描写モード（カメラ）の切り替え

別の視点から見たカメラを追加し，球体の動きをより分かりやすく観測できるようにする．

2.3.1　サブカメラの作成

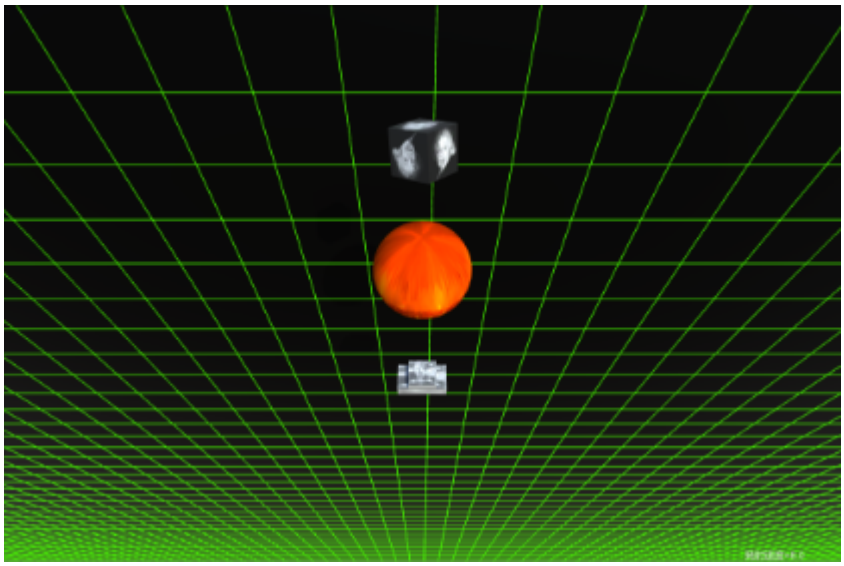

図 7: サブカメラ

カメラオブジェクトゲーム中に「2」キーを押すと図 7 のように視点が変更されるプログラムを作成する．まず，GameObject　→　Camera でサブのカメラオブジェクト Sub Camera を作成し，任意の位置.･向きを設定する. このカメラオブジェクトにスクリプトを実装し，public Camera で main と sub を宣言する.「2」キーを押したときの条件分岐で，Input.GetKeyDown(”2”) のとき，main.gameObject.SetActive(false); sub.gameObject.SetActive(true); とすることで main をオフにし，sub をオンに切り替える. これにより，物体の動きを別方向から観測することができ，立体に運動していることが分かりやすい. またメインカメラに戻すキーは「 1 」で設定する.

2.3.2　サブカメラとメインカメラの比較

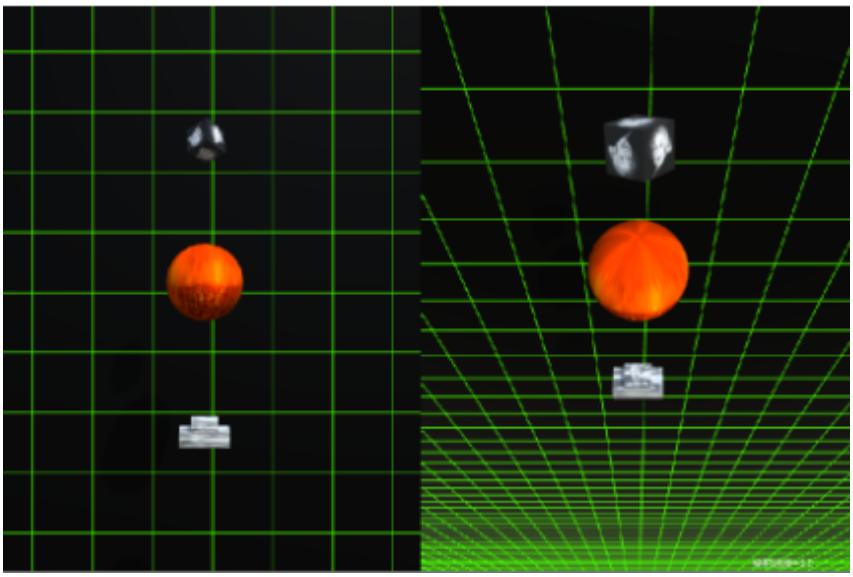

図 8: 比較カメラ

メインカメラとサブカメラを両方を観ることができる図 8 を，比較カメラとして「3」キーを押すことで切り替える変更されるプログラムを作成する.

条件分岐で，Input.GetKeyDown(”3”) のとき，gameObject.SetActive() を main,sub 共に true にする. 両方のカメラを表示するために，位置と大きさを，main.rect = new Rect(0.0f, 0.0f, 0.5f, 1f)，
sub.rect = new Rect(0.5f, 0.0f, 0.5f, 1.0f)　のように設定する.new Rect() 内の要素は順番に位置 x，y，大きさ横，縦である. この条件文を通った際，位置と大きさが変更されるため，別の条件分岐には，main.rect = new Rect(0.0f, 0.0f, 1.0f, 1.0f)，sub.rect = new Rect(0.0f, 0.0f, 1.0f, 1.0f)　と記載し，元に戻す.

2.3.3　球体視点のカメラ

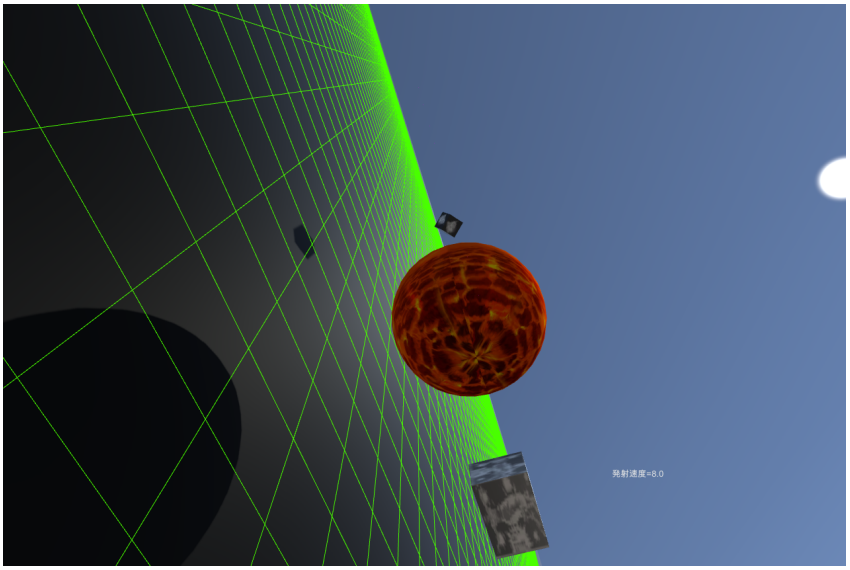

図 9: 球体のカメラ

図 9 は発射した球体から見たカメラである. 左クリックを押しながらマウスを動かすことで周りを見渡すことができる.「４」キーを押すことで切り替えることができる.

カメラオブジェクトを作成し，rigidbody を適応させ，それをプレハブ化させる．rigidbody は Kyu と同じ軌道にするため，Kyu の rigidbody と同じ値にする．2.2.2 章の球体を発射するスクリプトにカメラも発射するように追記していく，public Camera move で宣言し，プレハブ化したカメラを選択する．条件分岐 Input.GetKeyDown(”4”) で「４」キーが押されたとき，move.transform.position = hassya.position; で発射台の位置を指定する．ここで一度 rigidbody による影響をリセットするため，move.GetComponent<Rigidbody>().isKinematic = true; move.GetComponent<Rigidbody>().isKinematic = false; を記入する．

move.GetComponent<Rigidbody>().AddForce(transform.up * speed, ForceMode.VelocityChange); で直前に発射した Kyu と同じ速度でカメラを発射することができる．また 3 章で説明する，Kyu に適応させる万有引力のスクリプトをこの発射するカメラにも適応させておく．　発射後，マウス操作によって，周りを見渡せるように，newAngle.x -= (Input.mousePosition.y - lastMousePosition.y) * rotationSpeed.x，newAngle.y -= (lastMousePosition.x - Input.mousePosition.x) * rotationSpeed.y でクリック時の座標と現在値の差分値を求め，Camera.transform.localEulerAngles = newAngle で newAngle の角度をカメラ角度に格納する．

2.4　球体の軌跡の描写

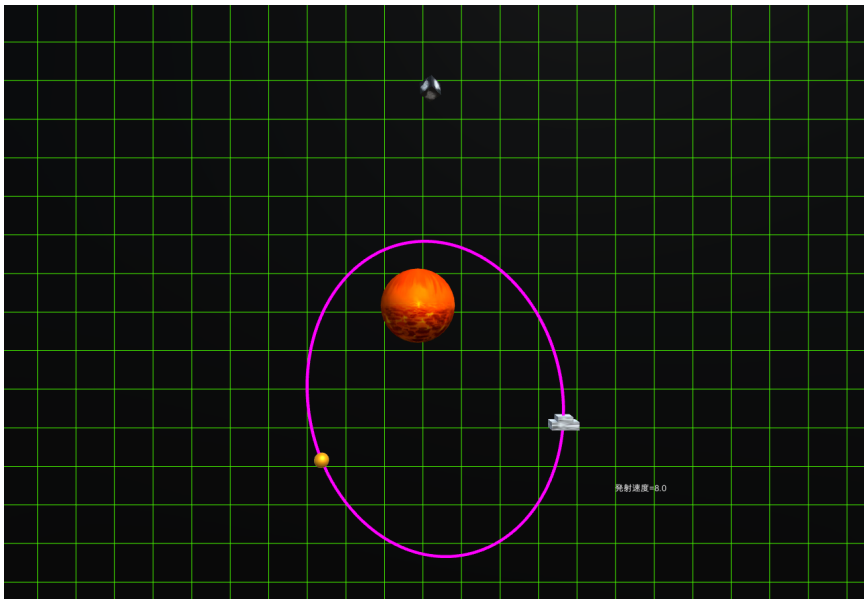

図 10: 球体の軌跡

惑星の引力で楕円運動する球の軌跡を描画する.Trail Renderer は，オブジェクトの位置を動かした時に，その軌跡を描くことができる. プレハブ Kyu に Add Component → Trail Renderer　で実装し，線の太さや描画し続ける時間を設定できる. これにより球体が正確に通った位置を把握することができる．図 10 は球を発射し，しばらく運動させたときの画像である．このように楕円運動していることが一目でわかる．

3　万有引力で動く物体実装

3.1　万有引力のプログラム

質量をもつすべての物体に万有引力が存在する．2 つの物体間の距離を r，2 つの物体の質量を m_1，m_2，万有引力定数をGとした力の大きさ F を，

$$F = G\frac{m_1 m_2}{r^2} \tag{1}$$

で与えられる．式 (1) は万有引力の法則である．これをプログラムで実装するにあたって障害物となるオブジェクト Sphere と kyu を式 (1) の関係となるスクリプトを作成する．
kyu は Sphere に引き寄せられるため，Sphere に向かう向き direction を，Sphere.transform.posotion - transform.posotion で取得する．Sphere と Kyu の距離は，direction.magnitude で取得でき，2 乗した値を distance とする．質量は GetComponent ＜ Rigidbody ＞ ().mass で取得する．最後に，GetComponent ＜ Rigidbody ＞ ().AddForce で太陽の方向に式 (1) の力を加えることができる．これらの処理を FixedUpdate() で固定インターバルで呼び出し，default では 0.02 秒毎に更新される．また図 10 では，kyu,Sphere の質量を 1，45 とし，kyu の初速度を 8，万有引力定数を 20 としている．

3.2　楕円運動の考察

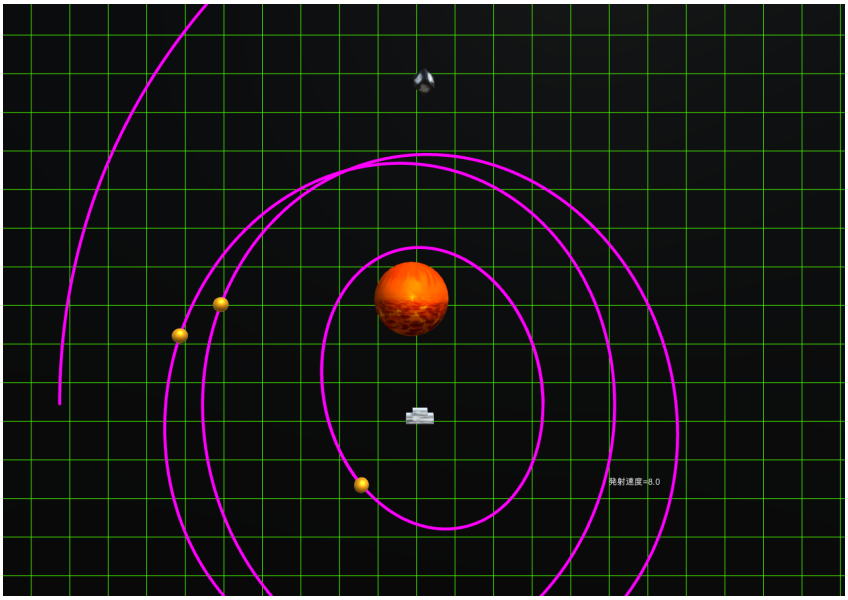

図 11: 複数の球体

図 11 は複数の球体を発射している図である．実際にシミュレーションした結果，Kepler の惑星の運動についての第 1 法則の通り，惑星は太陽を 1 つの焦点とする楕円運動を描いた．また楕円運動を観察していると，楕円の長径を通る速度が短径を通る速度より遅かった．Kepler の惑星の運動についての第 2 法則の太陽と惑星を結ぶ線分が単位時間に描く扇形の面積は，惑星それぞれについて一定であるためである．

3.3　２つの万有引力の考察

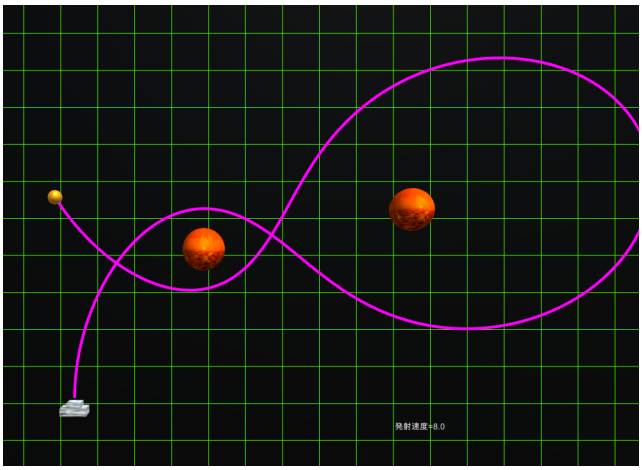

図 12: 2 つの惑星

図 12 は 2 つの惑星を固定し，重力源を 2 つにし，球体を発射させた図である．発射する位置によっては，図 12 のように八の字を描くことがある. これは，通常通り一つの惑星を楕円運動している途中で，もう一つの惑星との距離が小さくなり，働く力の向きが逆転したため，引き寄せられた結果であると考えられる．

3.4　5 次元世界モードの実装

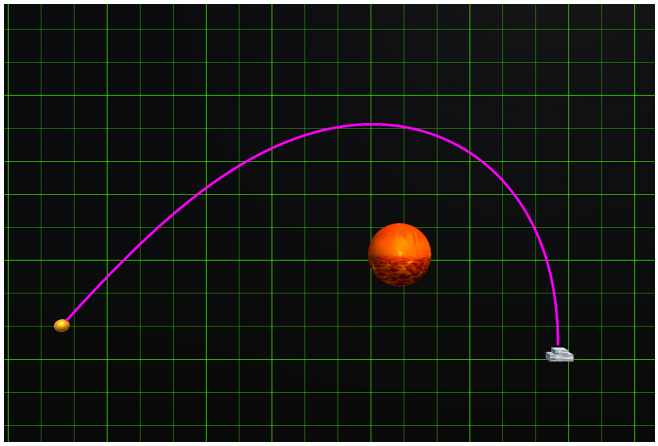

図 13: 5 次元世界の万有引力

$$F = G\frac{m_1 m_2}{r^3} \tag{2}$$

式 (2) は，式 (1) の距離 r を 2 乗から 3 乗にした．これにより 5 次元世界の万有引力を実装できる.「0」キーを押すと 5 次元世界モードに切り替わるようにする．

図 13 は 5 次元世界モードで球体を発射させた．通常の万有引力と違い距離の 3 乗で反比例させているため，重力源から離れると急激に万有引力が弱くなり，球体が放物線を描いた．

4　ブラックホールの実装

ブラックホールは通常の万有引力とは異なる力で実装する．擬ニュートンポテンシャルを使えば，重力ポテンシャルに似た運動方程式で，ブラックホール周辺の相対論的現象をうまく模倣することができる．

4.1　擬ニュートンポテンシャルのプログラム

式 (1) の距離 r からシュバルツシルト半径 a を引いた値を 2 乗し，

$$F = G\frac{m_1 m_2}{(r-a)^2} \tag{3}$$

により擬ニュートンポテンシャルを実装する．またブラックホールの質量を 200，半径を 2.0 とした．

4.2　ブラックホールを周回する光

ブラックホールを実装すると，ブラックホールの位置を確かめる手段が必要である．実際に発射して確かめることもできるが，ゲームとして球を無駄に消費するのではなく，事前に確かめられる機能を追加したい．ここではレーザーを発射してその曲がり具合でブラックホールの位置を特定できるような機能を作成する．

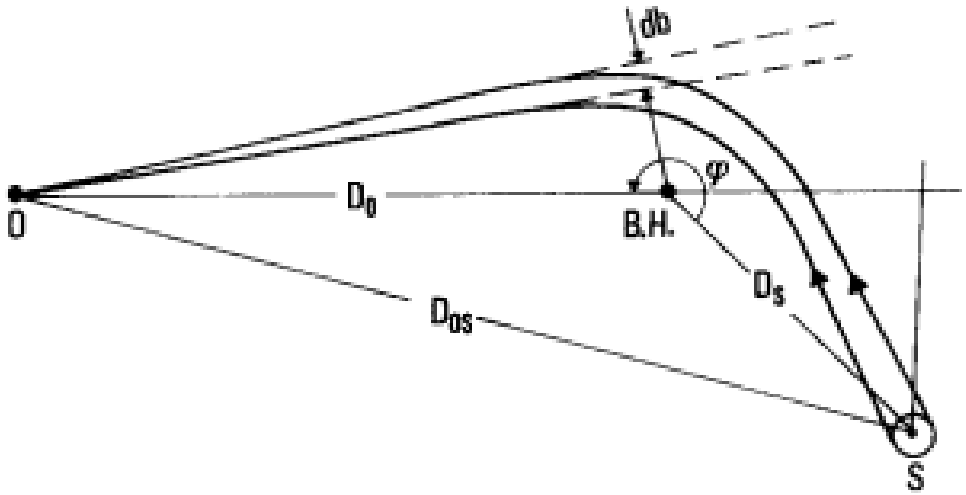

図 14: The black hole as a gravitational ”lens”（[1] 参照）

図 14 は，Hans.C.Ohanian のモデルの図で，b は衝突係数，O は観測者，$B.H.$ はブラックホール，S は光源，ϕ はブラックホールから光源を 0 度とした角度である．

4.3 光の軌道の計算方法

光の軌道を導出するには，次の運動方程式を用いる．

$$\frac{1}{b^2} - (\frac{du}{d\phi})^2 - u^2(1 - \frac{2GMu}{c^2}) = 0 \tag{4}$$

b は衝突係数 $u = \frac{1}{r}$ であり，r はブラックホールから光の間の距離を表している．また ϕ はブラックホールから光源までを 0 度とした，ブラックホールから光までの角度を表している．また，上記の式の $\frac{GM}{c^2}$ は定数で，$\frac{GM}{c^2} = 1$ として表すと，

$$\frac{1}{b^2} - (\frac{du}{d\phi})^2 - u^2(1 - 2u) = 0 \tag{5}$$

となり，この式を $\frac{du}{d\phi}$ の式として整理すると，

$$\frac{du}{d\phi} = \pm\sqrt{\frac{1}{b^2} - u^2(1 - 2u)} \tag{6}$$

となる. この式 (6) を $Runge - Kutta$ 法を用いて解く.

4.4　光の軌道の数値解析方法

1 解の微分方程式を解くための，$Runge-Kutta$ 法がある．本研究で光の軌道を実装するために使用する．以下は 4 次精度の $Runge-Kutta$ 法である．

以下の 1 階の微分方程式

$$\frac{dy}{dx} = f(x, y) \tag{7}$$

を考え，初期値を x_0，y_0，刻み幅を Δx とすると，計算方法は次のようになる．

$$k_1 = \Delta x f(x_n, yn) \tag{8}$$

$$k_2 = \Delta x f(x_n + \frac{\Delta x}{2}, y_n + \frac{1}{2}k_1) \tag{9}$$

$$k_3 = \Delta x f(x_n + \frac{\Delta x}{2}, y_n + \frac{1}{2}k_2) \tag{10}$$

$$k_4 = \Delta x f(x_n + \Delta x, y_n + k_3) \tag{11}$$

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{12}$$

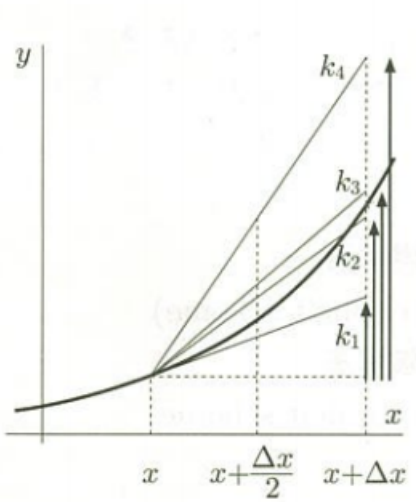

図 15: $Runge-Kutta$ 法 (4 次精度)

図 15 に示すように，x から $x + \Delta x$ の間での k_1，k_2，k_3，k_4 それぞれの値を重ねて平均したものを，y の値に加算することで次の y の値を求める．これを繰り返すことにより，解を導き出す方法である．

4.5　光の軌道の計算結果

光の軌道をシミュレーションするとき，観測者，ブラックホール，光源の位置をそれぞれ (-100，0)，(0，0)，(0，100) に置く，また，半径 2.0 の球対称のブラックホールとする．下の 3 つの図は，衝突係数のみを変えたときの比較である．

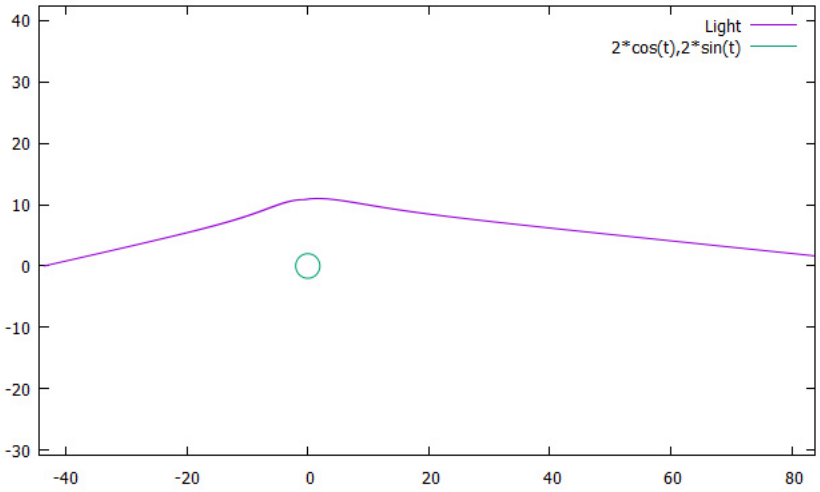

図 16: 衝突係数 b=10 の軌跡

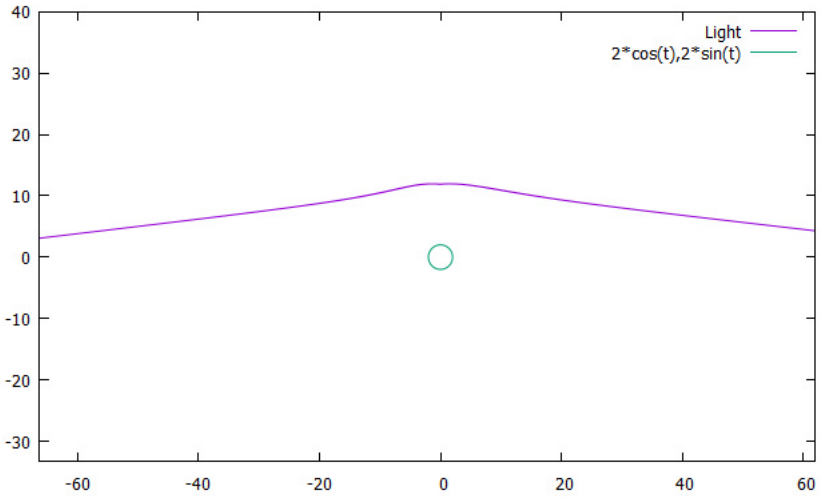

図 17: 衝突係数 b=11 の軌跡

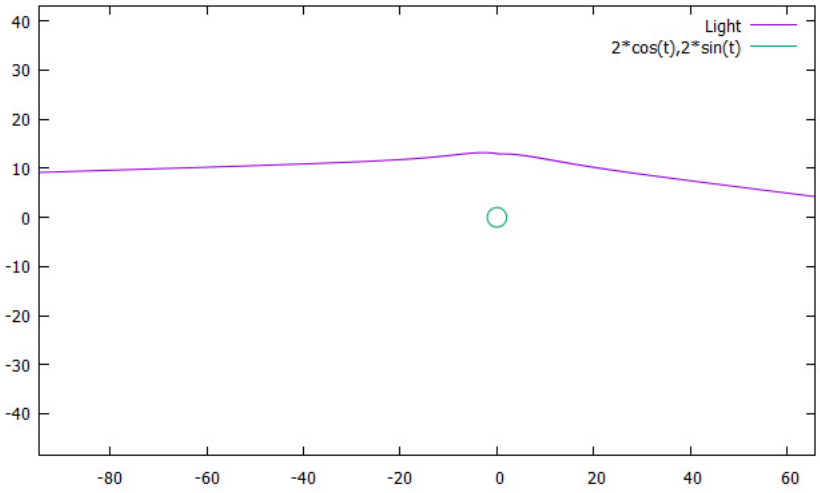

図 18: 衝突係数 b=12 の軌跡

図 16，図 17，図 18 の 3 つの衝突係数での結果を比較すると，衝突係数が低いほど光の曲がりやすいことがわかる．衝突係数が一番大きい図 18 では，他の図に比べてゆるやかであることがわかる．これは，衝突係数が大きいと，ブラックホールと光との距離が大きくなるため，ブラックホールの影響が小さくなるためである．

4.6　光の軌道の実装

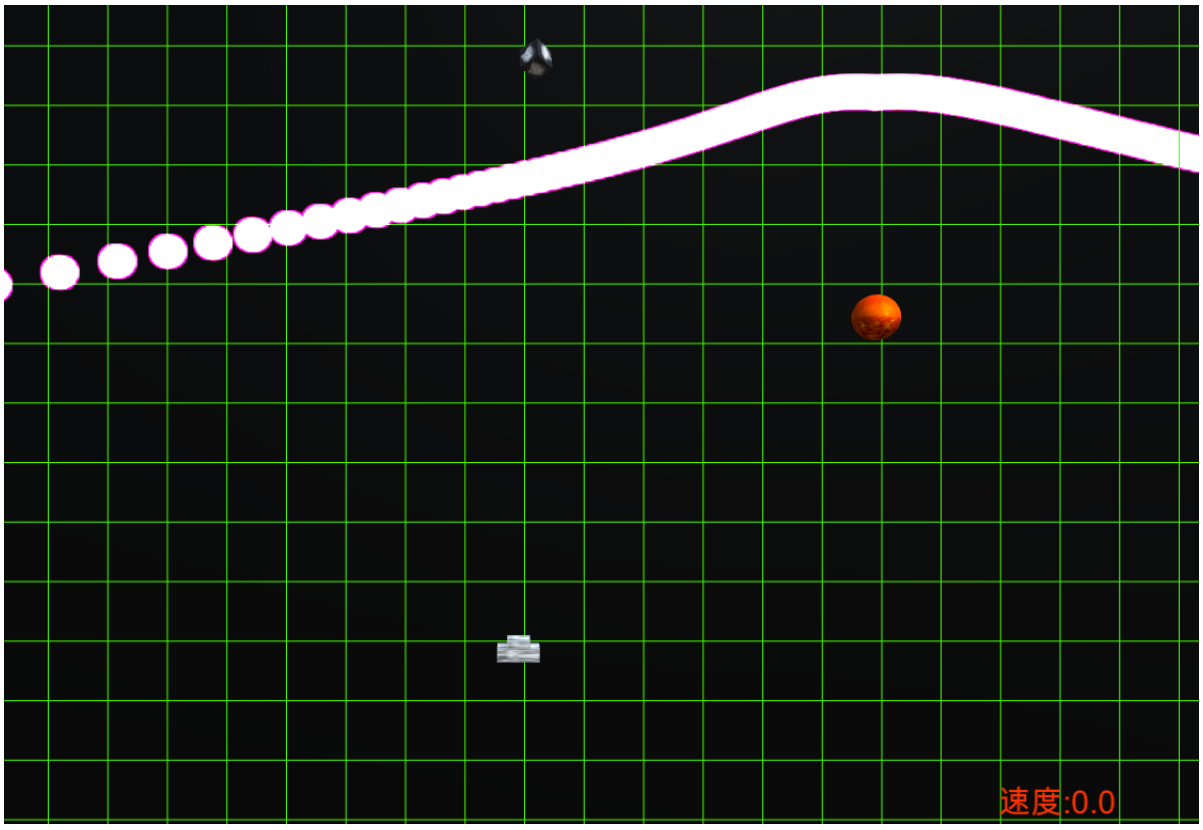

図 19: ブラックホール周りの光の軌跡

Unity でも $Runge-Kutta$ 法を用いて解いた. 光の実装は，GameObject → Effect → Particle System からエフェクトを複数配置して，一本の光のように見せた.「L」キーを押すとレーザーが発射される．図 19 は，説明のためブラックホールの位置に球体を配置しているが，実際のゲーム画面では表示されていない．刻み幅を 0.01 度，i を 0 から 18000 でループさせることで 0 度から 180 度までの光の軌道を描写する．このとき一つのループに一つエフェクトを描写すると 18000 個のエフェクトを同時に反映させることになる．処理の負担を抑えるために，i%500=0 で 500 ループ毎に描写すれば動作を軽くでき，見た目の影響も少ない．また光源と観測者の位置を，ブラックホールの位置 (x,y) を基準に x+50，x-50 とし，衝突係数を 8.5 とした．

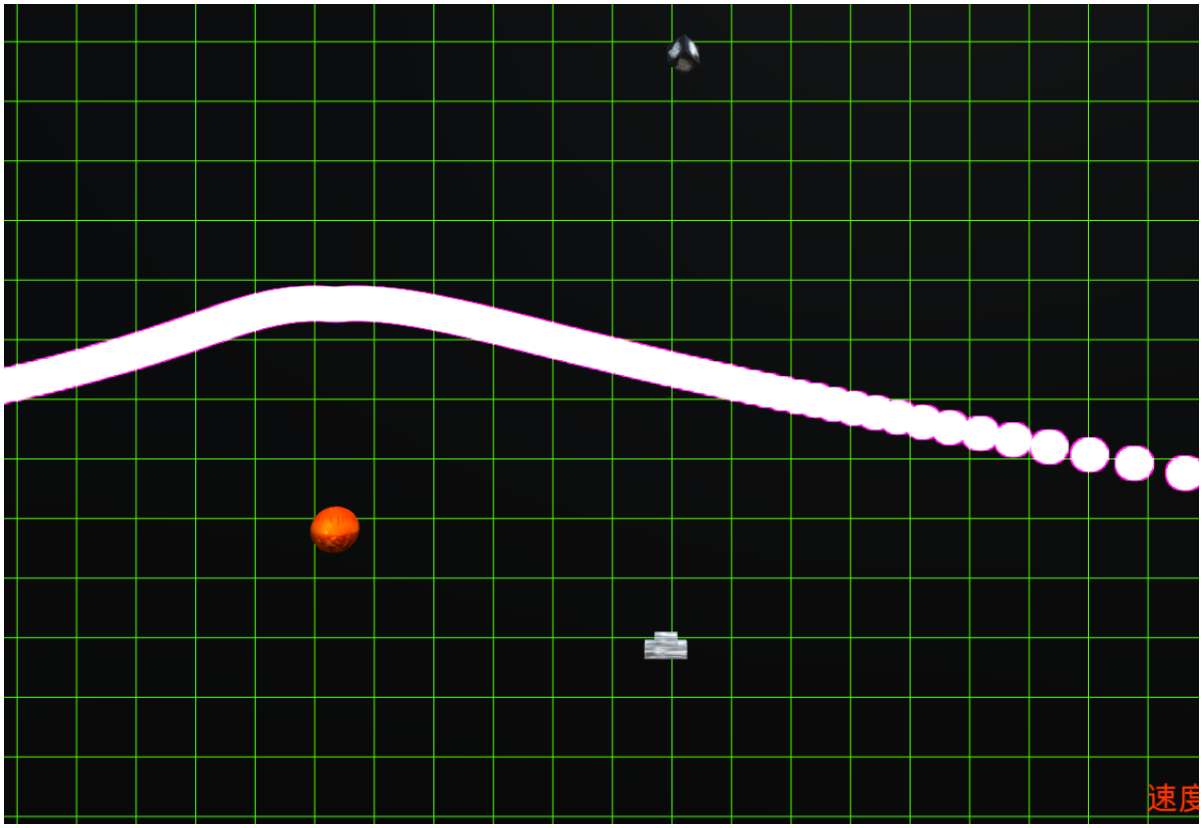

図 20: 目印のあるブラックホール周りの光の軌跡

図 20 は，図 19 からブラックホール位置だけを変更して「L」キーを押してレーザーを発射した時の図である．ブラックホールの位置によってレーザーの位置や形が変わり，プレイヤーがブラックホールの位置を推定できることがわかる．

5　まとめ

本研究で達成できたことを以下にまとめる．

・Unity を使って，Kepler の惑星運動の法則を満たすシューティングゲームを作成した．
・様々な視点から楕円運動を観察できるカメラを実装した．
・発射した球体が複数の万有引力を受けるステージを作成した．
・5 次元世界の万有引力を実装した．
・ブラックホールを実装し，レーザーの軌道でブラックホールの位置を推定できる機能を追加した．

今後の課題としてゲームとしての操作性や，ユーザーインターフェースを改善し，万有引力を応用したステージを追加して,『アインシュタインインベーダー』を Web で公開する予定である．

参考文献

[1] Hans.C.Ohanian,”The black hole as a gravitational ”lens””,Am.J.Phys.55(5),May 1987

[2] 真貝寿明,「徹底攻略　常微分方程式」