Тема 2.8. Алгоритмы управления камерой

План лекции:

2.8.1. Камеры в играх

2.8.2. Камеры в Babylon.js

2.8.3. Гравитация и столкновения (коллизия) в играх

2.8.1. Камеры в играх

Камеры в играх

Камера играет огромную роль в создании захватывающего геймплея. Управлять ей сложно, это требует большого опыта и креативности. Однако, если овладеть этим искусством, то повествование в играх можно вывести на новый уровень.

Давайте посмотрим, как положение камеры влияет на восприятие игрового мира.

Как располагать камеру в играх

Есть несколько основных типов расположения камеры, о которых стоит знать. Их можно встретить практически в любой современной игре.

Вид от первого лица

Такое расположение камеры позволяет игроку ощутить себя в роли персонажа.

В игре ты видишь только то, что видит герой: это помогает создать эмоциональную связь и заставляет сопереживать. В Outlast, например, игрок, как и персонаж, не видит, что находится за углом, поэтому эта игра довольно непредсказуемая (Рис. 1).

Рисунок1 — Игра «Outlast» (пример камеры от первого лица)

Встретить такую камеру можно в самых разных играх.

- **Платформеры:** Mirror’s Edge.
- **Шутеры:** Half-Life, Call of Duty, Far Cry.
- **RPG:** Skyrim, Oblivion, Deus Ex.

Вид от третьего лица

Эта позиция камеры делает игрока наблюдателем (Рис. 2).

Рисунок 2 — Игра «Dark Souls» (пример камеры от третьего лица)

Так игрок не только совершает какие-то действия, но и наблюдает за развитием истории. Он уже не воспринимает героя как самого себя, но всё равно может сопереживать ему.

Если камера далеко от персонажа, то он лучше видит обстановку, знает, где находятся враги или путь для отступления. Однако это ослабляет связь с происходящим.

И, наоборот, чем ближе персонаж, тем меньше обзор, но тем более близкой становится связь с персонажем. Однако, это мешает видеть всё, что происходит, поэтому приходится постоянно оглядываться, чтобы не дать застать себя врасплох. Особенно хорошо это работает в шутерах и хоррорах (и в шутерах-хоррорах).

Рисунок 3 — Игра «Resident Evil 4» (пример камеры от третьего лица)

Эффект усиливается, если приблизить камеру ещё сильнее. Особенно, если происходящее вызывает сильные переживания или страх.

Также такую камеру используют в симуляторах, например, в гонках.

Примеры игр с камерой от третьего лица:

- **Платформеры:** Prince of Persia.
- **Гонки:** Need for Speed, Burnout Paradise.
- **RPG:** Skyrim, Dark Souls, The Witcher.

Вид сбоку

Такой обзор также называют двумерным (Рис. 4).

Рисунок 4 — Игра «Mortal Kombat Komplete Edition» (пример камеры с видом сбоку)

Такая камера позволяет увидеть всё, что происходит в игре. Это важно для файтингов и платформеров, потому что в них игрок должен понять, откуда движется враг, как лучше добраться из точки А в точку Б, и так далее.

Здесь игрок выступает в роли зрителя и советчика. Только тут, в отличие от футбольных трансляций, персонаж охотно слушается.

Пример игр с камерой с видом сбоку:

- **Платформеры:** Super Mario, Terraria и многие другие.
- **Файтинги:** Mortal Kombat, Tekken.

Вид сверху

Такая камера позволяет ещё лучше видеть обстановку (Рис. 5).

Рисунок 5 — Игра «Diablo II: The Lords of Destruction» (пример камеры с видом сверху)

Это хороший приём для игр, где очень много врагов, за которыми постоянно нужно следить. Например, в Diablo количество NPC может быть огромным.

Такой обзор заставляет почувствовать себя вершителем судеб. Яркий пример — серия игр Sims (Рис. 6).

Рисунок 6 — Игра «The Sims 4» (пример камеры с видом сверху)

В играх вроде «Civilization» или «SimCity» игрок может почувствовать себя градостроителем или стратегом (особенно в стратегиях).

Примеры игр с камерой с видом сверху:

- **RPG:** Diablo, Torchlight.
- **Стратегии:** Warcraft, Starcraft, Command & Conquer.
- **Симуляторы:** Sims.

Динамичное положение камеры

Иногда разработчики выходят за рамки и отказываются от фиксированного положения камеры. Например, в «Mafia II» обычно камера находится сзади персонажа, но во время поединков она смещается вбок (Рис. 7).

Рисунок 7 — Игра «Mafia II» (пример динамичного положения камеры)

Меняя ракурс или расстояние между камерой и объектом, разработчики могут управлять ощущениями игрока, чтобы в нужный момент передать определённое чувство, эмоцию или настроение.

Например, в некоторых играх камера приближается, когда персонаж попадает в замкнутое пространство. Это помогает передать игроку чувство скованности и дискомфорта (Рис. 8).

Рисунок 8 — Игра «Uncharted 5» (пример динамичного положения камеры — приближение)

И, наоборот, на больших локациях камера отдаляется, чтобы игрок увидел масштаб и почувствовал, насколько мал его персонаж.

Рисунок 9 — Игра «Uncharted 5» (пример динамичного положения камеры — удаление)

Иногда камеру используют, чтобы сделать игру более кинематографичной. Например, в старом «Resident Evil 2» игрок не управлял камерой, поэтому разработчики могли в нужный момент добавить интриги или нагнать страха, изменив ракурс обзора.

Эффекты камеры

Чтобы сделать игру зрелищной, можно добавить разные эффекты для камеры. Самое простое — движение. В «Mortal Kombat Komplete Edition» камера перемещается, чтобы оба бойца всегда были в кадре.

Во время стрельбы можно добавить дрожь или отскоки, чтобы лучше чувствовалась отдача.

Чтобы показать скорость, в гонках добавляют размытие по краям экрана. Но такой эффект можно использовать и в других играх. Создатели «Outlast» решили его попробовать (Рис.9).

Рисунок 9 — Игра «Outlast» (эффект размытия по краям экрана)

Обзор «смазывается», когда игрок быстро поворачивает мышь, поэтому он всегда остаётся в напряжении, опасаясь не заметить приближающуюся угрозу.

Все эти приемы делают игру живее, зрелищнее и, следовательно, более захватывающей.

Камера и персонаж

Подобрать удачное расположение камеры непросто. Но это только полдела. Есть задача ещё сложнее: укрепить с помощью камеры эмоциональную связь игрока с персонажем.

В играх от первого лица можно видеть руки и иногда ноги персонажа (Рис.10).

Рисунок 10 — Игра «Skyrim» (видна рука персонажа в кадре)

Они воспринимаются игроком как собственные, и это помогает почувствовать себя метким стрелком или умелым фехтовальщиком.

В других играх персонаж иногда смотрит в камеру. Такой зрительный контакт создаёт эмоциональную связь. Например, в «Tomb Raider» героиня часто оглядывается — как бы на игрока. Это отличный способ показать её эмоции (Рис. 11).

Рисунок 11 — Игра «Tomb Raider» (взгляд персонажа в камеру)

Важно и то, что остаётся за кадром — например, звуки. Громкость звука должна зависеть от того, как близко к его источнику находится камера. В «Outlast» персонаж начинает учащённо дышать, когда ему страшно. От этого и игроку становится не по себе.

2.8.2. Камеры в Babylon.js

Камеры в Babylon.js

Существует широкий спектр камер для использования с Babylon.js. Наиболее часто используются две из них: **Universal Camera** и **Arc Rotate Camera**, а также **камеры виртуальной реальности**.

Рассмотрим их подробнее:

- **Universal Camera (Универсальная камера)** — подходит для шутеров от первого лица и работает с клавиатурами, мышью, сенсорными экранами и геймпадами. С помощью этой камеры вы можете проверять столкновения (коллизии) и применять гравитацию.
- **Arc Rotate Camera (Орбитальная камера)** — действует как спутник на орбите вокруг цели и всегда указывает на целевую позицию.
- **Follow Camera (Преследующая камера)** — подходит, например, для вида от третьего лица. Для работы с этой камерой нужно описать за каким объектом внутри нашей сцены мы хотим следить.
- **Anaglyph Camera** (Стереоскопическая камера) — предназначена для использования с 3D-очками.
- **Device Orientation Cameras (Камеры ориентации устройства)** — предназначены для реагирования на наклон устройства (например, смартфона) вперед или назад, влево или вправо.

Также существуют несколько камер, которые используются для создания игр в виртуальной реальности, но в данный момент мы не будем их рассматривать.

Так как камера является объектом (как и почти все в JavaScript), то она имеет свои свойства (ключи) и методы (функции объекта).
Давайте разберем более подробно некоторые свойства камер, а также способы их создания.

Universal Camera

Это лучший выбор, если вы хотите иметь в своей сцене управление, подобное FPS (Шутер от первого лица). Данная камера прекрасно взаимодействует с джойстиками.

Создание Universal Camera:

// Параметры : name, position, scene

```
var camera = new BABYLON.UniversalCamera("UniversalCamera", new
BABYLON.Vector3(0, 0, -10), scene);

// Направляет камеру в конкретную позицию. В данном случае в центр сцены

camera.setTarget(BABYLON.Vector3.Zero());

// Прикрепляет камеру к нашему Canvas (то где происходит отрисовка всего,
что мы видим)

camera.attachControl(canvas, true);
```

Arc Rotate Camera

Эта камера всегда указывает на заданный нами игровой объект (Mesh) и вращается вокруг него. Ей можно управлять с помощью стрелочек на клавиатуре, мыши или с помощью тачпада на ноутбуке.

Чтобы лучше понять то, как работает эта камера, представьте ее как о спутник, который вращается вокруг Земли. В данном случае Земля — наш Mesh, а камера — спутник (Рис. 12). Положение камеры относительно Mesh’а («Земли») можно задать тремя параметрами:

- alpha (продольное вращение в радианах).
- beta (широтное вращение в радианах).
- radius (расстояние от цели).

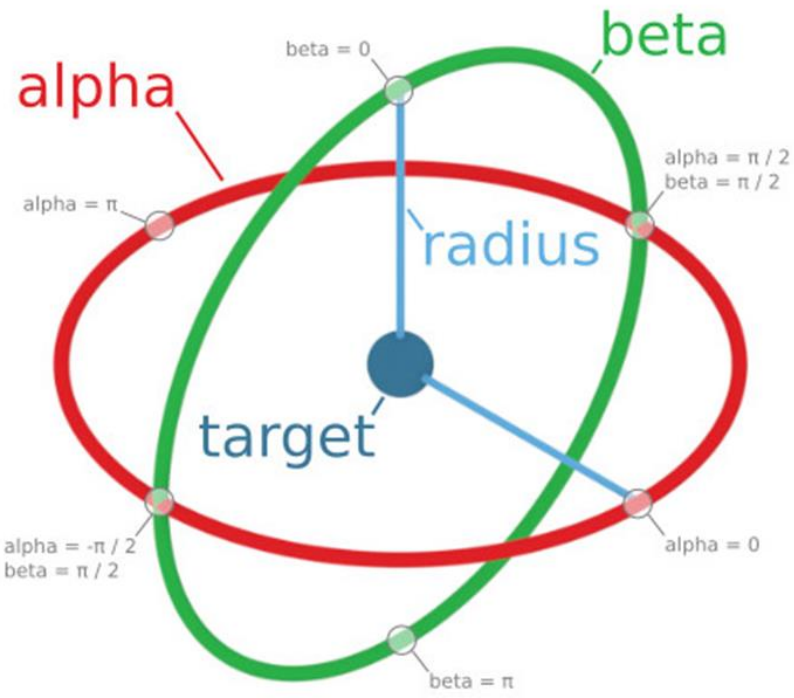

Рисунок 12 — Схема работы Arc Rotate Camera

Примечание: установка **beta** на 0 или PI может вызвать проблемы, поэтому в этой ситуации **beta** автоматически смещается на 0,1 радиана (около 0,6 градуса).

Обратите внимание, что **beta** увеличивается по часовой стрелке, а **alpha** — против часовой стрелки.

Давайте рассмотрим интересные свойства, которые есть у данной камеры:

ZoomToMouseLocation — если установлено значение **true**, колесико мыши будет увеличивать или уменьшать масштаб относительно текущего местоположения мыши, а не установленного нами Mesh’а в свойстве **Camera.target**. Это позволяет легко исследовать все уголки большой сцены.

WheelDeltaPercentage — если установлено ненулевое значение, то величина масштабирования будет установлена в процентах от радиуса камеры. Это означает, что масштабирование замедляется, когда мы приближаемся к Mesh’у, что здорово, потому что это означает, что мы можем более тщательно изучить свой объект вблизи.

Создание Art Rotate Camera:

// Параметры : name, alpha, beta, raduis, target position, scene

var camera = new BABYLON.ArcRotateCamera("Camera", 0, 0, 10, new BABYLON.Vector3(0, 0, 0), scene);

// Позиционирует камеру в новое место, обновляя значения, которые мы указали выше

camera.setPosition(new BABYLON.Vector3(0, 0, 20));

// Прикрепляет камеру к нашему Canvas (то где происходит отрисовка всего, что мы видим)

camera.attachControl(canvas, true);

Follow Camera

Follow Camera (Преследующая камера) делает ровно то, что заложено в ее названии — преследует цель. Достаточно лишь задать в свойство объекта **Camera.target** нужный нам объект, и камера переместится в целевое положение, из которого можно увидеть требуемый Mesh. Когда цель, к которой прикреплена камера, движется, то **Follow Camera** движется вместе с ней.

Начальное положение **Follow Camera** задается при ее создании, а целевое положение (то место, куда нужно переместиться) задается тремя параметрами:

- **radius**: расстояние от цели.
- **heightOffset**: высота над целью.
- **rotateOffset**: угол в градусах вокруг цели в плоскости XY.

Рассмотрим пример:

// Параметры: name, position, scene

var camera = new BABYLON.Follow Camera("FollowCam", new BABYLON.Vector3(0, 10, -10), scene);

// Расстоянии от камеры до нашего Mesh'a

```
camera.radius  = 30;

// Данное свойство показывает как высоко камера будет находиться над Mesh'ем

camera.heightOffset = 10;

// Угол вокруг нашего Mesh'a (игрового объекта)

camera.rotationOffset = 0;

// Ускорение камеры при движении от начальной позиции до Mesh'a

camera.cameraAcceleration = 0.005;

// Максимальная скорость перемещения камеры

camera.maxCameraSpeed = 10;

// Прикрепляет камеру к нашему Canvas (то, где происходит отрисовка всего, что мы видим)

camera.attachControl(canvas, true);

// Устанавливает за каким объектом (Mesh'ем) мы будем следить

camera.lockedTarget = targetMesh;
```

2.8.3. Гравитация и столкновения (коллизия) в играх

Гравитация и столкновения (коллизия) в играх

На этом занятии мы будем имитировать одни и те же движения камеры: камера перемещается на полу и может столкнуться с любыми объектами на сцене.

Определение и применение силы тяжести

Первое, что нужно сделать, это определить наш вектор гравитации.

Сцены Babylon.js имеют свойство гравитации, которое можно применить к любой камере, которую вы ранее определили в своем коде. Направление (да,

в Babylon.js мы можем сделать боковую гравитацию!) и скорость гравитации задается через **Vector3**.

```
scene.gravity = new BABYLON.Vector3(0, -0.15, 0);
```

Чтобы применить гравитацию сцены к камере, нужно задать для свойства **applyGravity** значение **true**:

```
camera.applyGravity = true;
```

В реальном мире гравитация — это сила которая направлена вниз, то есть в отрицательном направлении вдоль оси Y. На Земле эта сила составляет примерно 9,81 м/с². Падающие тела ускоряются по мере падения, поэтому требуется 1 секунда, чтобы полностью достичь этой скорости, затем скорость достигает 19,62 м/с через 2 секунды, 29,43 м/с через 3 секунды и т. д. В атмосфере сопротивление ветра в конечном итоге соответствует этой силе, и скорость перестает увеличиваться («конечная скорость»).

Однако Babylon.js следует гораздо более простой гравитационной модели — **scene.gravity** — представляет собой постоянную скорость, а не силу ускорения, и она измеряется в **единицах/кадр**, а не в **метрах/секунду**. По мере рендеринга каждого кадра, камера, к которой мы применяем гравитацию, будет двигаться по значению вектора вдоль каждой оси (обычно X и Z равны 0, но вы можете иметь «гравитацию» в любом направлении!), пока не будет обнаружено столкновение.

Итак, если вы хотите создать гравитацию, похожую на земную, вам нужно будет сделать некоторые предположения о количестве кадров, отображаемых в секунду, и вычислить подходящий вектор:

```
const assumedFramesPerSecond = 60; // Желаемое кол-во кадров в секунду - 60

const earthGravity = -9.81; // Гравитация Земли

scene.gravity = new BABYLON.Vector3(0, earthGravity / assumedFramesPerSecond, 0);
```

Поскольку все вычисления происходят один раз за кадр, камера на самом деле не «движется», она делает крошечные «прыжки» вдоль направления вектора гравитации. Это может быть важно, если вы полагаетесь на обнаружение столкновений, чтобы определить, «вошла» ли камера (или, скорее, Mesh, прикрепленный к ней) или «вышла» из какого-либо другого Mesh’a (например, плоскость под вашим «наземным» слоем, чтобы почувствовать падающего в пропасть персонажа и сбросить игровой

процесс). В зависимости от выбранной вами гравитации, начальной высоты, а также положения и высоты «триггерного» Mesh’а камера может прыгать прямо через Mesh, никогда не «пересекая» его.

Определение эллипсоида

Следующим важным шагом является определение эллипсоида вокруг нашей камеры.
Эллипсоид — это вытянутая сфера. Этот эллипсоид представляет размеры нашего игрока: событие столкновения будет возникать, когда другой Mesh вступает в контакт с этим эллипсоидом, предотвращая слишком близкое приближение нашей камеры к этому Mesh’у (Рис. 13).

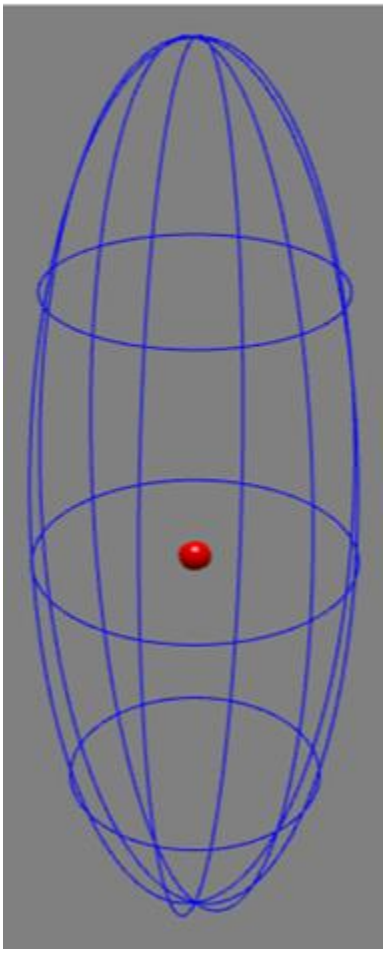

Рисунок 13 — Эллипсоид

Свойство эллипсоида на камерах Babylon.js по умолчанию имеет размер (0,5, 1, 0,5). Изменение этих значений сделает вас выше, больше, меньше, тоньше,

в зависимости от отрегулированной оси. В приведенном ниже примере мы сделаем эллипсоид нашей камеры немного шире и глубже, чем по умолчанию:

```
// Установим эллипсоид вокруг камеры (предположим, что это размер игрока)

camera.ellipsoid = new BABYLON.Vector3(1, 1, 1);
```

Обратите внимание, что эллипсоид для камеры немного смещен вверх, чтобы всегда иметь точку обзора поверх эллипсоида. Этим поведением можно управлять, обновив свойство **ellipsoidOffset**.

Применение коллизии

Наш последний шаг — начать реагировать на столкновения на нашей сцене:

```
scene.collisionsEnabled = true; // Включаем возможность коллизии внутри сцены

camera.checkCollisions = true; // Включаем коллизию отдельно для камеры
```

А также устанавливаем всем объектам нашей игры, с которыми мы хотим сталкиваться, свойство **checkCollisions** в **true**:

```
ground.checkCollisions = true; // Коллизии землей (чтобы ходить по земле)

box.checkCollisions = true; // Коллизии с коробкой
```

Управление камерой

Управлять и двигать камерой легко! Каждая камера в Babylon.js автоматически обрабатывает клики мышью или нажатия на клавиатуру, как только вы вызываете метод **attachControl** у камеры. Можно забрать возможность управления с помощью метода **detachControl**. Рассмотрим пример:

```
myCamera.attachControl(canvas);
```

Заключение

Сегодня мы рассмотрели различные камеры в Babylon.js. Камера — это основа любой игры, ведь от того, как игрок смотрит на мир, зависят впечатления, которые он получит.