Chess game animation in blender

Jake Moss - s46409665

June 8, 2021

Contents

1 Introduction

1.1 Project aims

This project aims to demonstrate a sufficient knowledge of computer graphics techniques and implementations through the creation of a visually appealing chess game animation tool. This was accomplished using Blender, and its python scripting API.
This project utilises two tools to create the chess animation

- Blender
  Blender had a large appeal due to is its extensibility through `Python` for this project The exposed API allows its users to script typical actions, and develop add-ons in a familiar and standard environment. Part of this scripting API allows the user to add and remove objects, insert key-frames, and change the properties of an object, anything a user can do with a mouse and keyboard, is able to be configured programmatically. This allows Blender to be used a front end to any `Python` or `C++` program.

- `python-chess` library
  The `python-chess` library is a chess library for python with move validation, generation, and `PGN` (Portable Game Notation, the most common file format for chess games) parsing. This library is designed to be able to function as back-end, making it perfect to use in conjunction with Blender.

2 The plan

During the proposal the plan was to create an interactive, real time chess board, however this was changed as the techniques and concepts would be severely limited. Blender with python scripting was a perfect compromise.

3 Blender implementation

3.1 Modelling, textures, and shading

The scaling of the models was deliberately chosen to be unrealistic in order to simplify the translation from position with the back to front end (See Python side - Array index to world space)

3.1.1 Nodes

Blender nodes allow for the creation of textures and shaders through a pipeline of simple operations to create complex procedural results. Its simple to understand visual workflow is a popular alternative to layer based compositing. **node-vs-layer**
Throughout this project, procedural texture and shading generation using nodes was used instead of the traditional texture wrapping using UV mapping in order to give objects consistent and appealing surfaces. This provides two relevant benefits;

- Tweaking textures and shading doesn't require an any more work than changing values on the respective node.

- Textures can be applied to any model without fitting issues, i.e. repetition and resolution.

3.1.2 Shaders

A shader is a program, typically run on the GPU, to computer the colour of a group or individual pixel. These shaders describe the lighting interactions of objects or surfaces, such as reflection, refractions, and absorption.

3.1.2.1 Principled BSDF

A `BSDF` (Bidirectional Scattering Distribution Function) describes how light scatters on a surface. In computer graphics, computing a highly detailed microsurface is not feasible, instead it is replaced with a simplified macrosurface (See Figure 1). As the surface no longer retains the detail it would in reality, light behaves differently on this new macrosurface. To compensate for this a `BSDF` is used that matches the aggregate direction scattering of the microsurface (at distance). **ggx-paper**

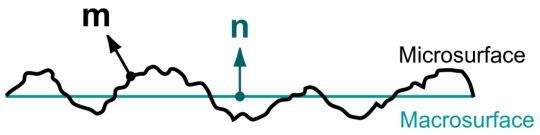


Figure 1: Micro vs macro surface (Source)

Blenders implementation breaks down this problem down into two separate functions by assuming that the microsurface can be adequately described using a microfacet distribution function and a shadowing-masking function.
Blender provides these options for the distribution and shadowing functions, and the subsurface methods.

- Distribution[1]
  - GGX
    `GGX` is a `BRDF` (bidirectional reflection distribution function) which aims to be a faster than its alternative `Multiple-scattering GGX` at the cost of physical accuracy. The `MDF` describes the distribution of microsurface normals **m** (Figure 1) while the shadow masking function describes what fraction of the microsurface normals **m** are visible. **ggx-paper**

    In `GGX` the shadow masking function does not account for reflections or scattering. This can create excessive darkening and a loss in energy conservation in some areas **principled-bsdf-docs**.
  - Multiple-scattering GGX
    Almost all popular parametric `BSDF`'s only consider single reflection to account for self-shadowing but omit outgoing light that scatters multiple times between

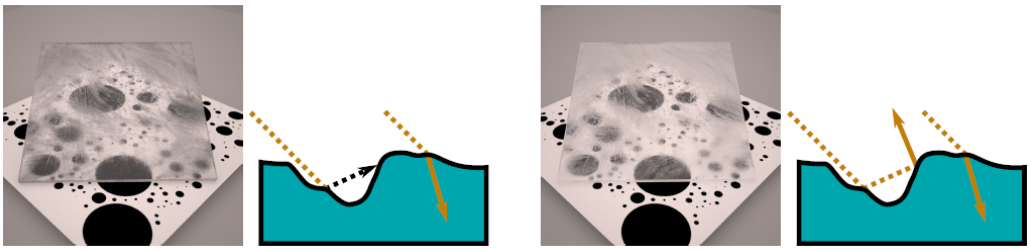


Figure 2: Single scattering (left), multiple scatters (right), (Source)

microfacets. Omitting outgoing light breaks conservation of energy and leads to dark patches within rough surfaces **ms-ggx-paper**.

Blenders `Multiple-scattering GGX BRDF` allows for multiple light bounces within microfacets to achieve 100% energy conservation and provide a more physically accurate render **principled-bsdf-docs**, **ms-ggx-paper** . It accomplishes this by conducting a random walk on the microsurface until the ray escapes. Unlike `GGX` there is no known analytical expression for this model (Blender's specific implementation), it must instead be solved stochastically **blender-issue-tracker**. This comes at a performance cost, the original papers cites a 19% penalty using a Monte Carlo physically based renderer, Blenders development forums estimates the performance penalty to be approximately 3% at the time of implementation **blender-issue-tracker**.

- Subsurface Scattering Method Subsurface scattering when light passes through an object that is normally opaque. Described by a `BSSRDF` (bidirectional subsurface scattering reflectance distribution function)

  – Christensen-Burley
  The `Christensen-Burley` method is an approximation of a physically based volumetric scattering system with faster evaluation and efficiency **Christensen-Burley**.

  – Random walk
  Apposed to the approximations use the in the `Christensen-Burley` model, the `Random walk` modelling uses true volumetric scattering inside the mesh. Due to this the model does not perform well when the mesh is not closed. This accuracy comes at a cost of rendering time (actually performance hit is largely dependent on the model itself), and increased noise.

  Accuracy within the subsurface scattering was not an area of importance within this report thus the `Christensen-Burley` model was chosen due to its better performance.

All renders within this report have `Multiple-scattering GGX` enabled as the benefit was need to outweigh the cost.
The `Principled BSDF` shader is a combination of multiple layers into a single node. This is done for ease of use.

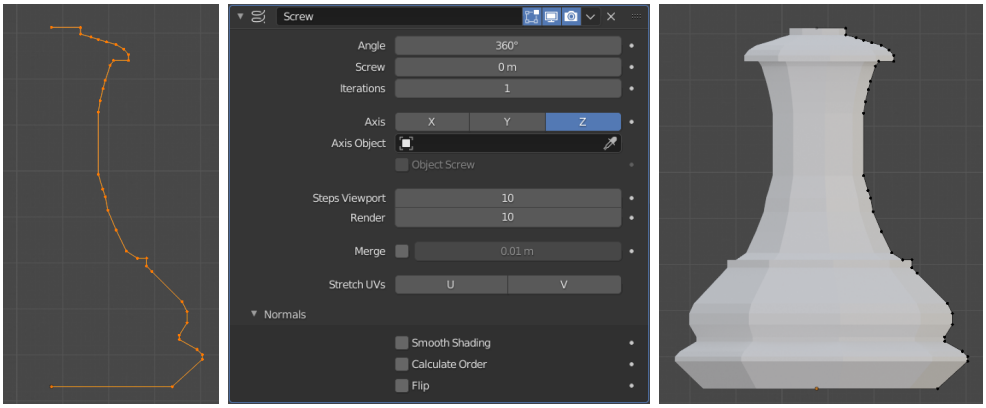
[1]Note that the `Distrubution` option Blender gives is different from the `Microfacet Distrubution Function`, and includes both the `MDF` and the Shadow-masking function.

This shader encapsulates bidirectional reflectance and transmittance distribution functions. Individually these functions determine how light behaves on the surface and inside a material.

3.1.3 Pieces

Pieces were modelled after the reference image below Figure 3. From this image the pieces where traced using the `Add Vertex` tool, from the Add Mess Extra Objects add-on. To transform this line of vertices to a solid object a `Screw` modifier was applied.

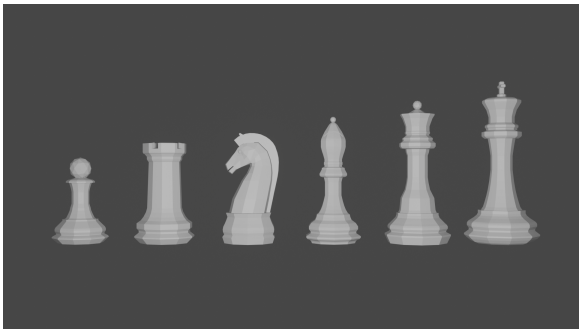
Figure 3: Reference image, Licensed under Pixabay License



The notable changes from the default settings is the lowering of the steps from 16 $\rightarrow$ 10 and disabling `Smooth Shading`. This was a stylistic choice as it was believed that the low polygon look would better demonstration reflections and the planned indirect lighting (See Lighting - Disco Ball).

To model the knight, 3 separate reference images where used. The base was constructed in a similar manner to the other pieces. The head was modelled manually (read painstakingly). Additionally ico-spheres where added to piece some pieces additional detail. The finally piece models appear as below.

Figure 4: Knight reference images, (Source)



3.1.4 Board

3.1.4.1 Chess board

The chess board model is a simple rectangular based prism with dimensions `8m x 8m x 0.4m`. The checker board texture comes from the `Checker Texture`, with `scale=8.0` and black and white colours. This texture output is then feed into the base colour input of a `Principled BSDF` shader node.

Within world space the board was positioned in the positive, positive quadrant such that the very bottom left handle corner of the board was at `0,0` with each squares dimensions as `1m x 1mx`. This positioning becomes important in Python implementation - Array index to world space.

3.1.4.2 Marble exterior

The marble exterior was added to showcase reflections, shadows, bloom, and specular highlights through the use of procedural texture and normal mapping.

To accomplish this texture layered Perlin noise was used. The first set of switch derives its coordinators from the `Texture`

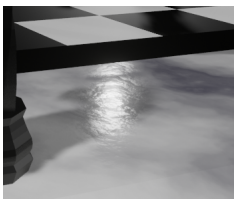


Figure 6: Marble normal texture

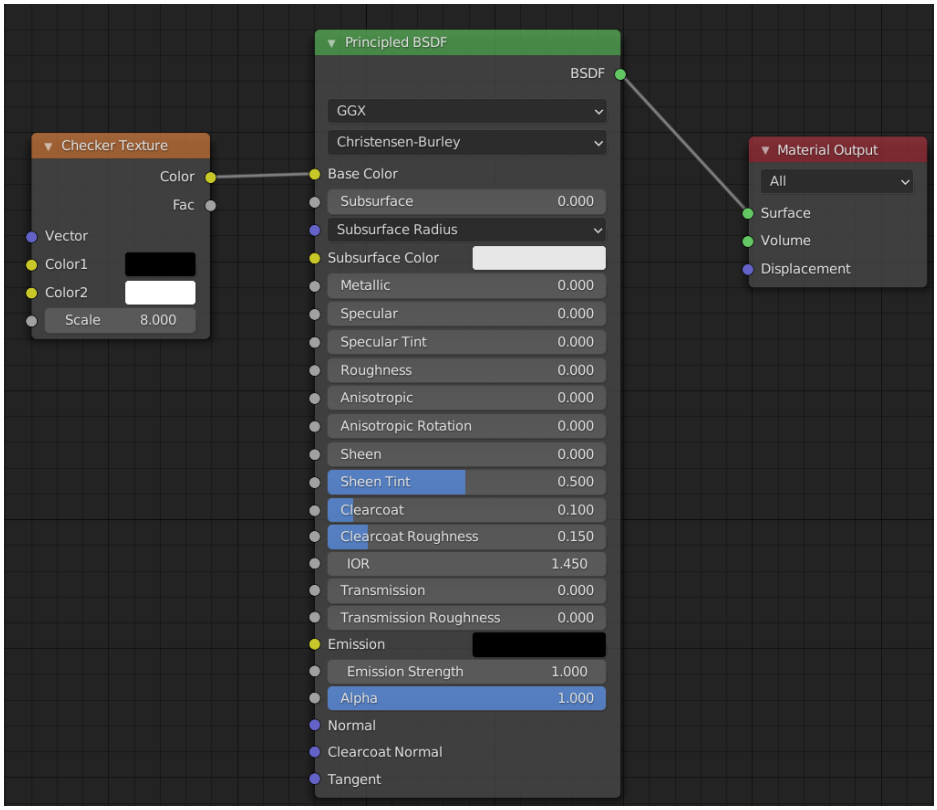


Figure 5: Complete checker board texture.

`Coordinate` node which is set to use the mesh [2]. The `Mapping` node is extraneous in this example, simply used to move the Perlin noise map around. The second layer of Perlin noise uses the already nosy surface to create the dark patches marble typically feature. The colour ramp is used to brighten and shade the darker patches of noise. This colour is then used as the base colour for the `BSDF` shader. To give this texture some depth the colour output of the ramp is used to create a normal map. Colour data (yellow node) is used a vector input can yield unexpected results. To remedy this the colour data is first past through a `Bump` node as the height data and then fed into the `BSDF` shader.

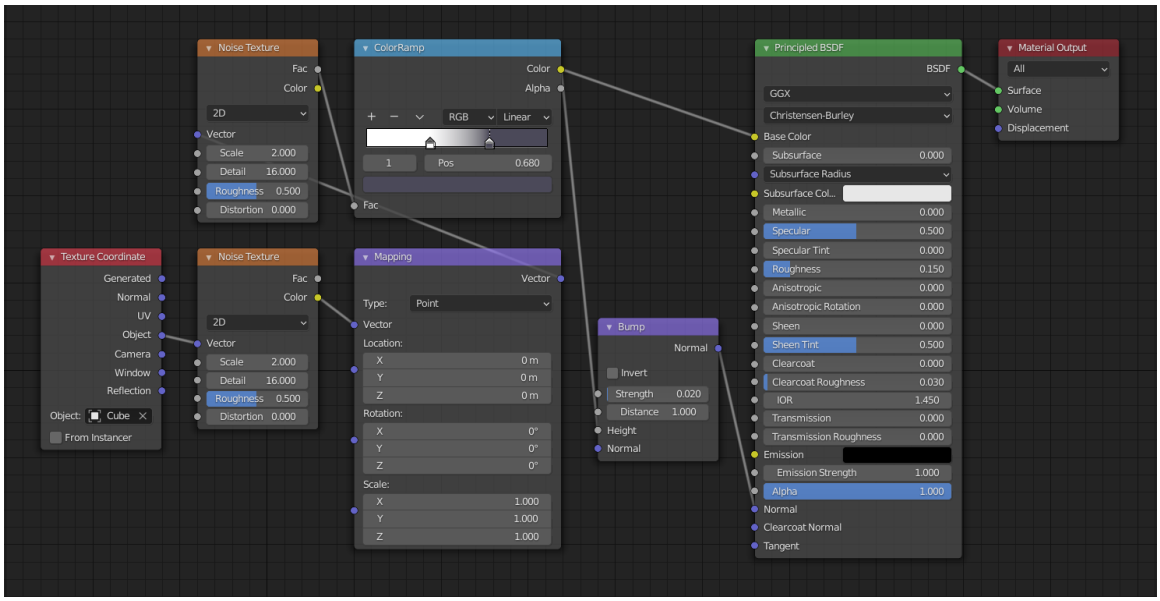


Figure 7: Complete marble texture

[2] A very interesting effect and be achieve by using the `Camera` as the source of the coordinates. The marble texture flows like sea foam as the camera spins.

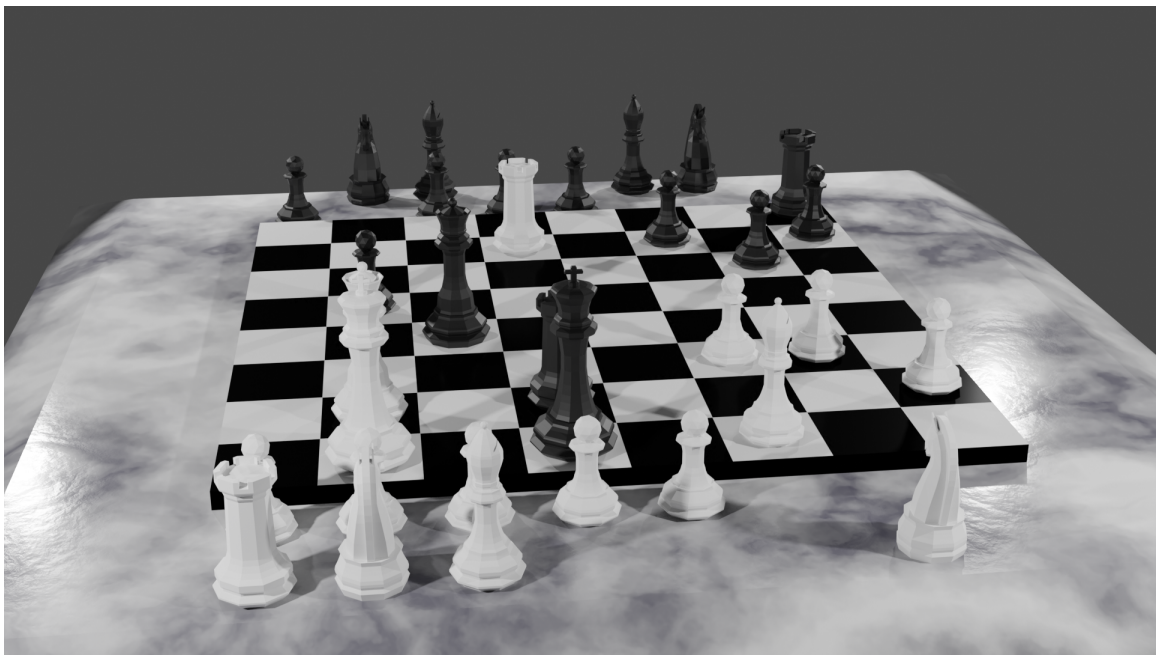


Figure 8: Cycles marble render

Figure 9: Eevee marble render

3.2 Particle effects

3.2.1 Explosions

Initially an explosion was planned an each piece capture as a way to add some flare. However this was quickly scrapped as the additional baking and rendering time were deemed obscene. A demo render featuring a smoke cloud was created and can be viewed here master/Videos/smoke.mp4.

3.2.2 Confetti

As an alternative to explosions, a confetti shower on the winning king (only checkmates deserve confetti) was added instead.

The confetti is still an explosion but with the outgoing particles are set to a collection of "confetti's". The source of the explosion is a upwards facing dome, this gives the confetti a even spread. The particles are set to have randomised, size, rotation, angular velocity, normal velocity. To achieve the slow fall effect the effective gravity of the particle simulation was lowered to 38% its normal strength. Each confetti has a texture with `Roughness 1.0` and varying base colours.

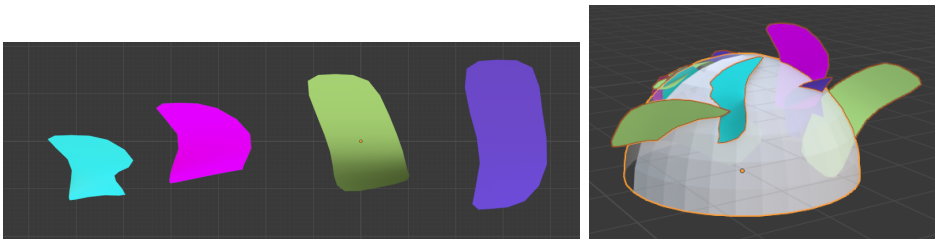


Figure 10: Confetties (left), confetti source (right)

A sample render can be seen below.

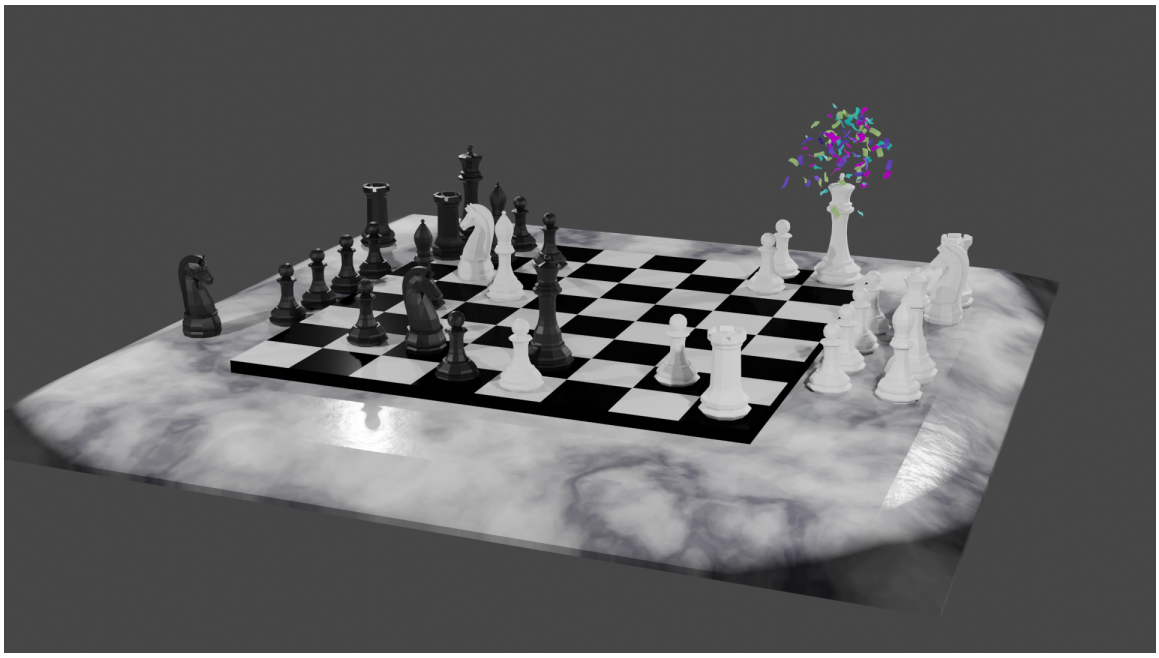


Figure 11: Confetti render using Cycles

3.3 Lighting

The power of Blender lights is measured in `Watts`, however this is not the same wattage that consumer lights bulbs are rated in. Blenders light power is rater in radiant flux, which is the measure of radiant energy emitted per unit time apposed to consumer light bulb which are rated in lumens, or luminous flux **radiant-flux**, **luminous-flux**. Luminous flux differs from radiant flux in that luminous flux is adjusted to the varying sensitivity of the human eye. **luminous-flux**

Table 1: Blender light power conversion (Source)

Real world light	**Power**	**Suggested Light Type**
Candle	0.05 W	Point
800 lm LED bulb	2.1 W	Point
1000 lm light bulb	2.9 W	Point
1500 lm PAR38 floodlight	4 W	Area, Disk
2500 lm fluorescent tube	4.5 W	Area, Rectangle
5000 lm car headlight	22 W	Spot, size 125 degrees

3.3.1 Direct

To directly light the scene four spot lights placed at the corners of the board were used. Spot lights were chosen instead of typical points lights or sun lights to give consistent directional lighting during the camera spins. The four lights are set to track the centre of the chess board. To account for the unrealistic scale the power of the lights was set to higher than

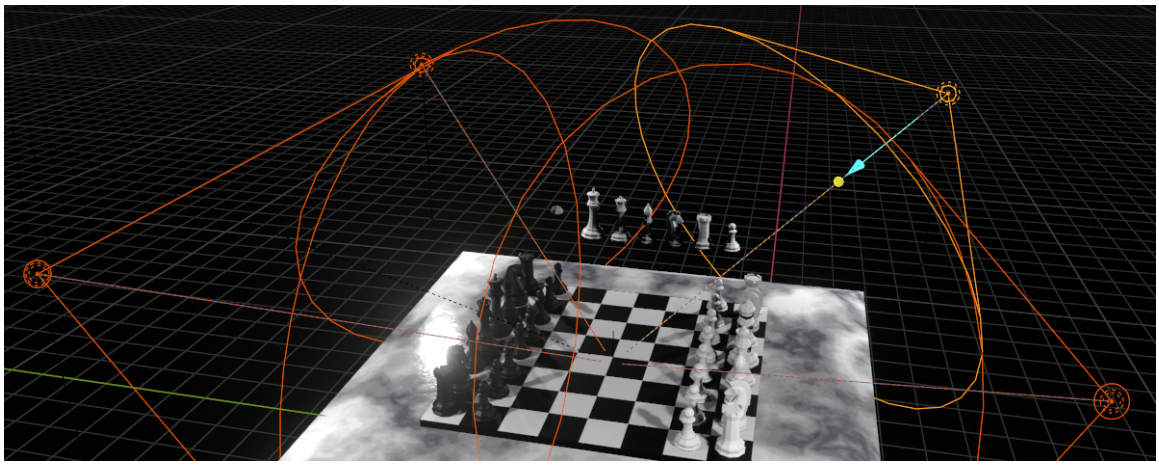


Figure 12: Spot light positions

normal. However, this was still not enough. To avoid cranking the lights to approximately `1MW` the exposure within the film settings was adjusted to compensate as per the Blender documentation **light-power-docs**.
The blend for this light was set to 0 as the cone fully encompasses the board.

3.3.2 Indirect

There is minimal indirect lighting within the final scene (all models reflect some amount of light due to their texturing, however this is not a significant amount). Considering this a second scene was created which featured considerable indirect lighting. To accomplish this a disco ball was implemented.

3.3.2.1 Disco ball

Whats better than chess? That's right, disco chess! There is no better way to demonstrate indirect lighting that a giant ball of mirrors in the middle of the board.
The disco ball model is a simple iso-sphere with a mirror like texture. Its rotation is achieved through simple key-frames.
To create a mirror in Blender the `Roughness` parameter on the `Principled BSDF` shader node is set to 0. This alone didn't make a very convincing disco ball, in addition to the `Roughness`, the following values were tweaked,

- `Metalic 1.0`
  This change made the ball most disco like as it gives a fully specular reflection tined with the base colour without diffuse reflection or transmission.

- `Specular 1.0`
  While the `Matallic` value is responsible for the majority of the specular reflection, this change gave the ball a halo like glow.

- `Base colour`
  With the two previous changes the disco ball appears too uniform and reflective, some

faces appear completely blown out with no variance between them. To remedy this a `Voronoi` noise texture's colour output was piped into a colour ramp (this is done to avoid the **very** pink look the `Voronoi` noise produces).

Figure 13: `Voronoi` texture colour output

A `Voronoi` noise texture is as procedural Worley noise function evaluated at the texture coordinate. The patterns are generate by randomly distributing points. From these points a region is extended, the bounds of this region is determined by some distance metric. The standard Euclidean distance is used here.

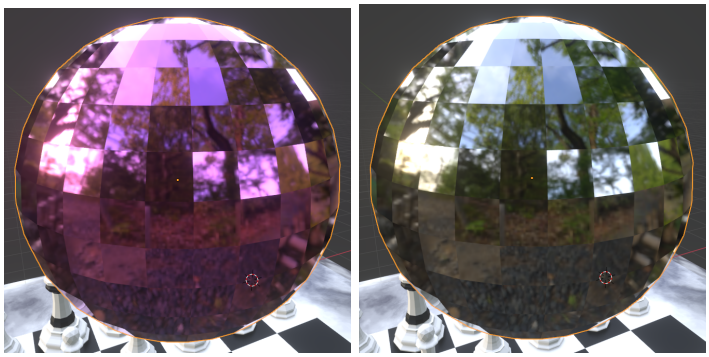


Figure 14: No colour ramp (left), complete texture (right)

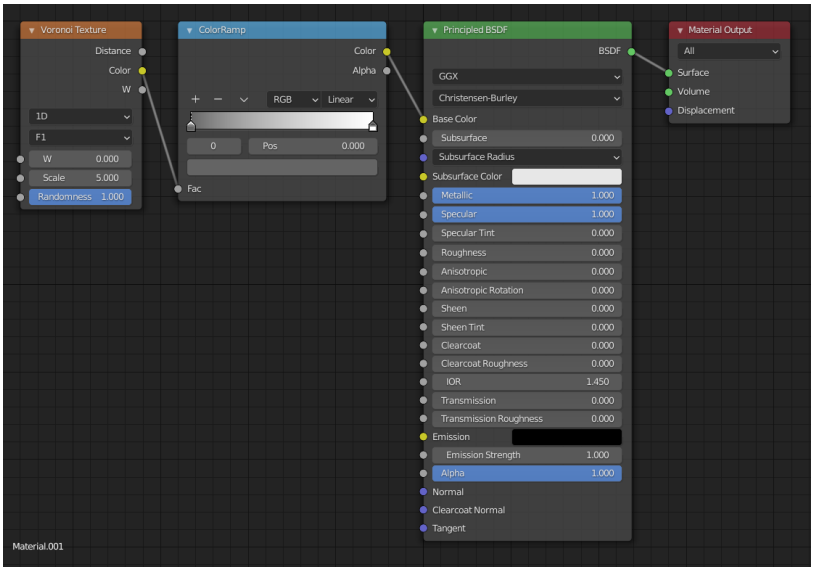


Figure 15: Complete disco ball texture

With the disco ball texture and model complete a two red and blue lights were trained on the disco ball. The results can be seen with the Render engine section.

3.4 Render engine

Blender offers two modern rendering engines, when working on a project it is important to keep in mind what engine is desired as to keep in mind the limitations of both. For this project `Eevee` was the chosen engine due to the significantly lower render times than `Cycles`, however renders using both engines were still made in order to compare them.

This project also made use of third engine, `Luxcore`. `Luxcore` is a free and open source rendering engine designed specifically to model physically accurate light transportation.

3.4.1 Eevee

Blenders `Eevee` is designed to be used within the view port for real time rendering previews `Eevee` must cut a lot of corners to achieve its speed. Although these approximations are physically based, the behaviour of light suffers. By default mirrors do not function (without explicitly enabling) and caustics are not present at all.

`Eevee`'s pipeline, while lacking signifcant documentation, is that of a typical rasterisation engine.

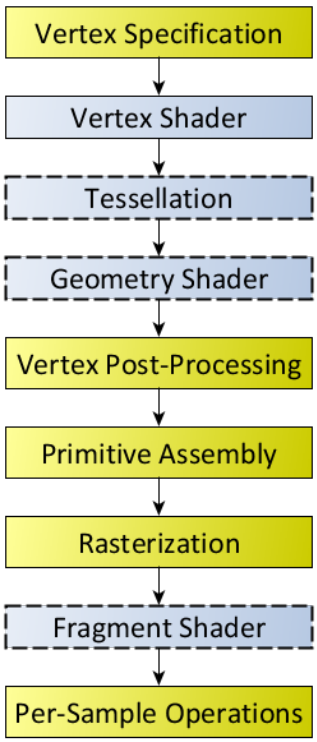


Figure 16: Rendering pipeline (Source)

1. Vertices are retrieved from the buffer object [3]. This includes vertex colours, UV coordinates, and vertex locations of polygons.
   (a) The retrieved vertices are transformed to the post-projection space by the vertex shader.
   (b) (Optional) Tesselation is where patches of vertex data are subdivided into smaller interpolated points. This is useful to dynamically add or subtract detail from a polygon mesh.
   (c) (Optional) The geometry shader is used to conduct layered rendering which is useful when cube based shadow mapping or rendering cube enrichment map without having to rend the entire scene multiple times.

[3] A buffer object is an array of unformatted memory allocated by the GPU.

2. Vertex post-processing - output from the previous stages is collected and prepared for the following stages.
   (a) In Transformation Feedback the output of the Vertex Processing stages is recorded and place into buffer objects, preserving the post-transformation rendering state such that it can be resubmitted to various processes.
   (b) Primitive Assembly prepares the primitives for the rasterizer by dividing them into a sequence of triangles and stored in an efficient method.
   (c) Geometry outside of the view is culled and vertices are transform from NDC to screen-space.
3. Rasterization projects the geometry onto a raster of pixels and outputs a collection of fragments. Each fragment represents a rasterized triangle. Multi-sampling occurs when multiple fragments come from a single pixel.
4. The fragment shader takes these fragments and computes the `z-depth` for each pixel along with the colour values. Colour is computed using the surfaces `BSDF`.
5. Per-sample operations are used to cull fragments that are not visible, and determine the transparency.

Outside of this pipeline (concurrently) are compute shaders, often these are used to compute arbitrary information, i.e. tasks not directly related to drawing triangles or pixels. Particles and fluid simulations, and terrain height map generation are common application of compute shaders.

3.4.2 Cycles

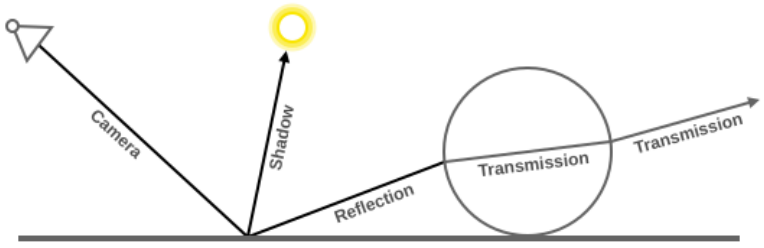


Figure 17: Light path diagram (Source)

`Cycles` is a physically based backwards path tracing rendering engine. While this engine is physically based, it is not physically correct. It does not aim to be either **design-goals**.

In backwards path tracers the paths are generated starting from the camera. They are bounded around the scene until encountering the light source they "originated" from. This is considered an efficient method for direct lighting as a ray will always yield some result opposed the forward path tracing where many light rays may never reach the camera.

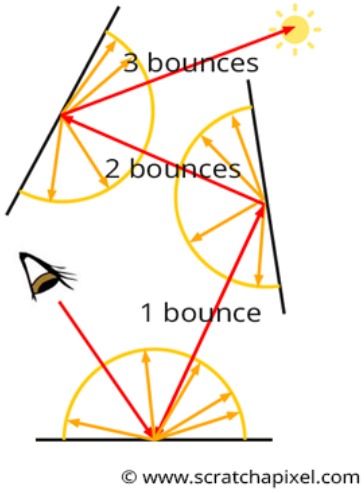


Figure 18: Backwards path tracer (Source)

Cycles offers two integrators, `Path Tracing` and `Branched Path Tracing`. Where `Branched Path Tracing` differs is that a pure path tracer, each bounce of the light ray bounces in one direction and will pick one light to receive lighting from, while a `Branched Path Tracing` will split the path for the first bounce into multiple rays and sample from multiple lights. Naturally this makes sampling considerably slower, however it offers lower noise in scenes primarily lit by a one or single bounce lighting. It is possible to split the ray on successional bounces as well however the complexity will increase exponentially for diminishing returns.

To combat noise the `OptiX` denoiser was employed as it operates best on NVIDIA hardware.

Being a jack of all trades Cycles suffers in some areas when compared to specialised engines such as `Luxcore`. Specifically Cycles suffers significantly in advanced light effects such as caustics as explicitly noted in their documentation **blender-sampling**.

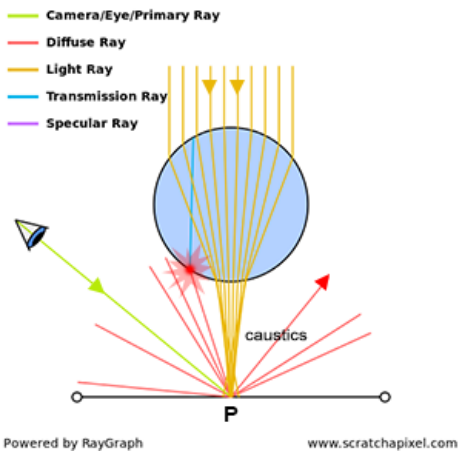


Figure 19: Caustics diagram (Source)

Caustics are the optical phenomenon of light patterns forming on surfaces through reflections

and refraction. These are notably difficult to calculate efficiently [4] using a unidirectional path tracer as many rays will not collide with the object that should be focusing light. One solution to this is a technique called photon mapping. In photon mapping an initial pass of the scene is done using a forwards path tracer to follow the path of light rays as they interact with glossy and refractive objects.

Cycles is significantly slower that `Eevee` (`1min vs 16 seconds` for some single frame renders). With adaptive sampling the number of samples conducted in less noisy areas is automatically resulting in a performance improvements.

3.4.2.1 Thank you to Jack

Due to significant hardware limitations for ray-tracing (`GTX 760, i5-4670`), a favour was called in with a good friend, Jack kindly lent their `RTX 2070` for a cycles render. See master/Videos/Marble_cycles.mp4.

3.4.3 Luxcore

Another algorithm to calculate caustics is bidirectional path tracing. For each sample two paths are traced independently using forward and backwards path tracing. From this every vertex of one path can be connected directly to every vertex of the other. Weighting all of these sampling strategies using Multiple Importance Sampling creates a new sampler that can converge faster than unidirectional path tracing despite the additional work per sample. This works particularly well for caustics or scenes that are lit primarily through indirect lighting as instead of connection rays to the camera or source, instead rays are connected to each other. This allows rays that are very close to form a path **Caustic-Connection**.

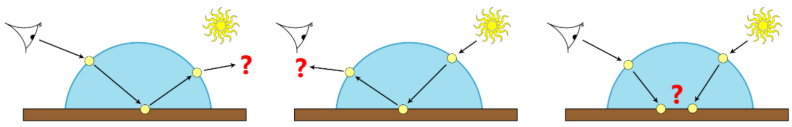


Figure 20: Different connection strategies (Source)

`Luxcore` documentation recommends enabling the `Metropolis sampler` when using bidirectional path tracing. This allows `Luxcore` to spend more time sampling bright areas of the image and thus rendering caustics with greater accuracy. However, this causes significantly more RAM usage. This was not enabled during the `Luxcore` render due to memory restrictions.
`Luxcore` offers caching of caustics through the `PhotonGI caustic cache` option however this is only applicable the scene this static (with the exception of the camera).

[4] There is nothing inherently difficult about caustics, any path tracer can render them. Instead it is a matter of convergance speed.

3.4.4 Tragedy - 22:20, 01/June/2021

At 10:20pm on the first of June the PC that had been enslaved to rendering a Luxcore for more than 96 hours straight, died. It had been a good 8 years, but she finally gave out. Official cause of death is unknown but it suspected to be something to do with power delivery.

A successful data recovery was conducted the next morning. Only the last 12 frames were missing, they were rendered on another device. See master/Videos/disco_luxcore.mp4

4 Python implementation

4.1 Processing games

Reading and stepping through games is handled almost entirely by the chess library. No special considerations need to be made here. The minium working example below demonstrates all that is necessary to step through an entire game.

```
import chess
with open(filename) as pgn:
    game = chess.pgn.read_game(pgn) # Parses pgn file
    board = game.board()

    for move in game.mainline_moves():
        board.push(move) # Pushs the move to the move stack, this "makes" the move
```

4.2 Pairing problem

During a game of chess there is nothing in between moves, simply one discrete board state after another. This is also how the chess library makes moves, by computing differences and tracking board states, while this is reliable and simple it does not play nice when games become continuous (animated).
Initially this script also tracked the board state using a dictionary, with the square as the key, and corresponding blender object as the value, pushing and pop at each move. However, this presented difficulties when implementing animations and special moves and animations. The code was generally cluttered and not up to an acceptable quality.

4.3 The solution

To remedy the mentioned problems a custom class was devised, and aptly name `CustomPiece`. This class acts as a generalised representation of a piece which is able to act upon itself and the Blender model it puppets. Stored within an unrolled 2d array with the index representing its position on the chess board (See Python implementation - Array Index to world space) the object is able to move itself within the array while handling move and capture animations. Special move handling is generalised into the main loop, (See Python implementation - Special moves).
This design approach has clear advantages such as

- Adheres to the `Model-View-Controller` design philosophy.
- Array and object manipulation is not handled at any higher level than required.

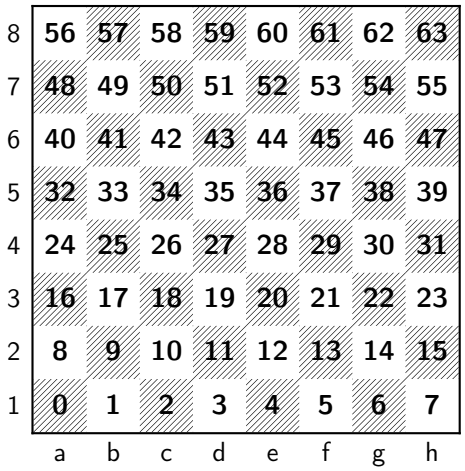
- Translation between the chess library interface and Blenders API is seamless.
- Creates a unique object that pairs a Blender model to a `python-chess` `PieceType`.

However, the self-referential nature of objects manipulating the array their are stored in adds significantly to the complexity. Luckily the implementation is simple.
An initial sketch of this class can be seen here 25.
Implementation can be see here 7.

4.4 Array index to world space

`python-chess` provides great functionality to retrieve what square a move is coming from, and going to. Internally this is stored as a `int` representing each square in 1d array notation.

```
Square = int
SQUARES = [
    A1, B1, C1, D1, E1, F1, G1, H1,
    A2, B2, C2, D2, E2, F2, G2, H2,
    A3, B3, C3, D3, E3, F3, G3, H3,
    A4, B4, C4, D4, E4, F4, G4, H4,
    A5, B5, C5, D5, E5, F5, G5, H5,
    A6, B6, C6, D6, E6, F6, G6, H6,
    A7, B7, C7, D7, E7, F7, G7, H7,
    A8, B8, C8, D8, E8, F8, G8, H8,
] = range(64)
```



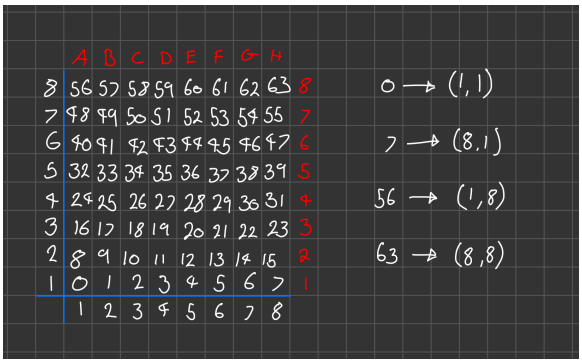


Figure 21: Array representation ((`tl`) Source code, (`tr`) Chess board, (`b`) Overlaid)

To convert form array indexing two simple expressions were used.

$$x = (\text{INDEX mod } 8) + 0.5$$

$$y = (\text{INDEX div } 8) + 0.5$$

[5] Note the addition of 0.5 is to centre the pieces on the board squares in world space and will be excluded from further examples.

4.4.1 Abuse of this functionality

While modulo will always produce a positive integer between $0 \rightarrow 7$, integer division can result negative numbers and is not bounded. Using this the mapping can be extended past the board it was designed for. This provides an easy method to place captured piece after their animation. By storing each pieces initial position, and adding or subtracting 16 depending on the colour, pieces can be placed 2 rows behind their initial position.
Two rows behind was preferable to the respective position on the other side of the board to avoid the inversion required so that the pawns would be in front of the back rank pieces.

72	73	74	75	76	77	78	79
64	65	66	67	68	69	70	71
56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7
-8	-7	-6	-5	-4	-3	-2	-1
-16	-15	-14	-13	-12	-11	-10	-9

Figure 22: Extended conversion

[5] Note `div` here is integer division.

4.5 Special moves

Figure 23 shows the main loop logic, used to move the correct pieces.

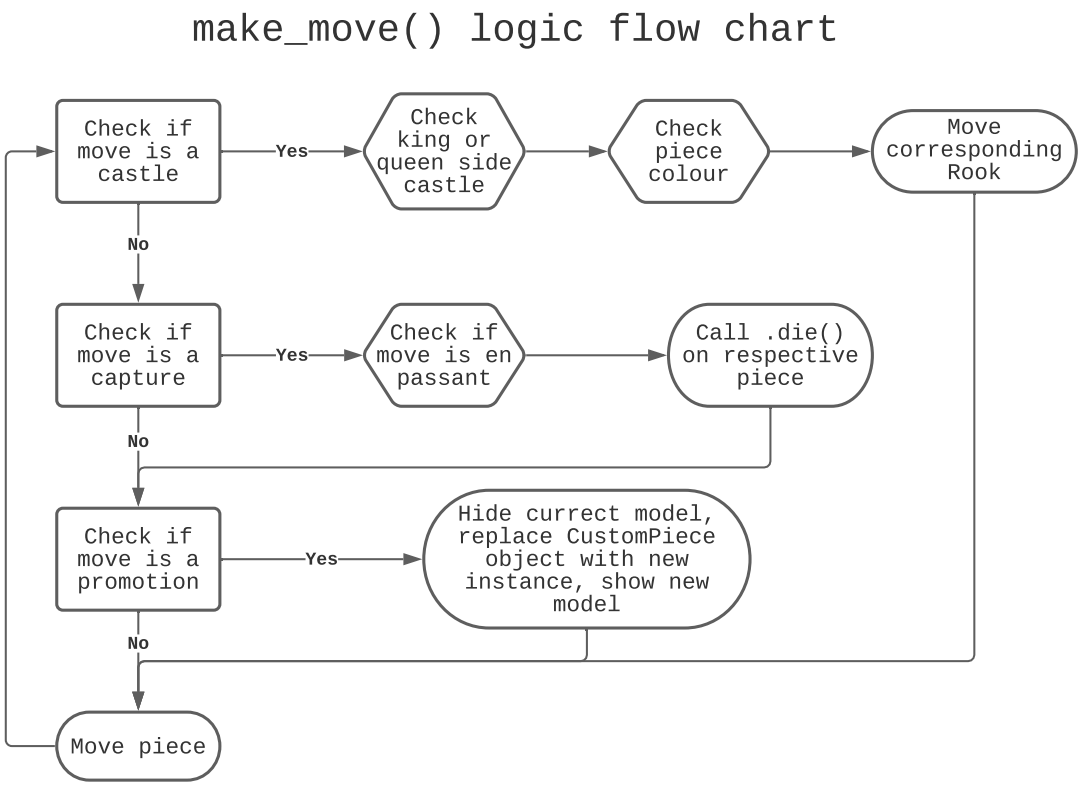


Figure 23: Main loop logic

4.5.1 Castling

Within standard chess there are only four castling possibilities, these are easy enough to check naively. This is the only section that limits this script to standard chess. To extend support to `chess960`, a bit-board mask of all the rooks with castling rights could be filtered to obtain the index of the rook that will be castled. See the documentation.

```
if board.is_castling(move):
    if board.turn: # White
        if board.is_kingside_castling(move):
            array[chess.H1].move(chess.F1)
        else: # queen side
            array[chess.A1].move(chess.D1)
    else: # Black
        if board.is_kingside_castling(move):
            array[chess.H8].move(chess.F8)
        else: # queen side
            array[chess.A8].move(chess.D8)
```

4.5.2 En passant

The `python-chess` library makes handling en passant a breeze. The move is checked if it is an en passant first, then as only one square is possible of an en passant on any move that position is retrieved.

```
    else: # standard case
        if board.is_capture(move):# is en passant, great...
            if board.is_en_passant(move):
                array[board.ep_square].die() # NOTE, object is gc'ed
            else: # its a normal capture
                array[locTo].die() # NOTE, object is gc'ed
```

4.5.3 Promotion

Contained within a separate conditional is the promotion logic. This is handled separately from the rest of the logic as a move can be both a capture and a promotion.

```
    array[locFrom].move(locTo) # NOTE, piece moves always

    if move.promotion is not None:
        array[locTo].keyframe_insert(data_path="location", index=-1)
        array[locTo].hide_now() # hide_now unlinks within blender
        pieceType = move.promotion # piece type promoting to
        array[locTo] = CustomPiece(chess.Piece(pieceType, board.turn),\
                                   SOURCE_PIECES[chess.piece_symbol(pieceType)],\
                                   array, locTo) # shiny new object
        array[locTo].show_now()
```

A new key-frame is inserted initially as the piece that will promote has already been moved and that animation needs to finish before it can be hidden.
Within the Blender view port the pieces that will be promoted too already exist at the right position, they are just not rendered until needed.

4.6 Animation

4.6.1 Key frames

To animate an object within blender two key-frames must be inserted with different values for some property at varying times. Blender will then interpolate between them (See Python implementation - Interpolation for interpolation methods)
Key-frames for all pieces are inserted every move. This is done to ensure stationary pieces stay stationary. Every move the piece has 10 frames to complete its moving animation. Between each move there a 3 buffer to provide some separation between moves.
In addition to piece animations, the camera also rotates at a rate of 2° per 13 frames.

```
        FRAME_COUNT = 0
        keyframes(array) # intial pos
        FRAME_COUNT += 10
        for move in game.mainline_moves():
            scene.frame_set(FRAME_COUNT)

            make_move(board, move, array)
```

```
            keyframes(array) # update blender

            camera_parent.rotation_euler[2] += radians(2) #XYZ
            camera_parent.keyframe_insert(data_path="rotation_euler", index=-1)

            board.push(move) # update python-chess

            FRAME_COUNT += 10
            keyframes(array) # update blender
            FRAME_COUNT += 3
```

While the camera's rotation is tired to the length of the game, in order to continue spinning while the remaining animations (confetti and captures) finish additional key frames are added. Confetti is conditionally added to the winning king. No confetti for a draw.

```
        confetti = bpy.data.collections["Board"].objects['Confetti source']
        if board.outcome() is not None:
            winner = board.outcome().winner
            king_square = board.king(winner)
            xTo, yTo = square_to_world_space(king_square)
            confetti.location = Vector((xTo, yTo, 3))
            bpy.data.particles["Confetti"].frame_start = FRAME_COUNT
            bpy.data.particles["Confetti"].frame_end = FRAME_COUNT + 12

        print(FRAME_COUNT)
        for _ in range(5):
            scene.frame_set(FRAME_COUNT)
            camera_parent.rotation_euler[2] += radians(2) #XYZ
            camera_parent.keyframe_insert(data_path="rotation_euler", index=-1)

            FRAME_COUNT += 13
```

In order to move the camera with a fixed rotation and radius from the centre of the board the camera was made a child of a `Empty Plain Axis`. Rotations and translations applied to the camera parent are also applied to the camera. This allows for ease fixed distance rotations.

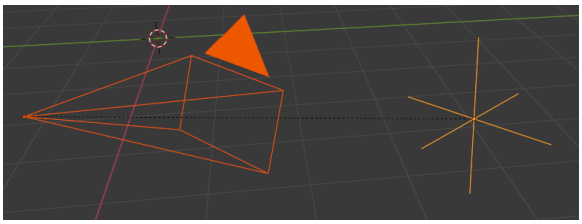


Figure 24: Camera parent axis

4.6.2 Interpolation

Blender offers 3 curves for interpolation between key-frames.

- Constant
  Object value only objects on the last possible frame.

- Linear
  Object value has a changes linear between the key-frames to form piecewise continuous curve.

- Bézier
  The object value is interpolated using a Bézier curve. Bézier curves are parametric curves used in computer graphics to create smooth surfaces, or in this case, a smooth function between two points.

  Blender implements a forward differencing method for a cubic Bézier curve evident from the source code **blender-source**.

By default Blender uses Bézier curve interpolation for all motions. This is the preferred option for piece movement. However, linear was opted for the camera motion although a cubic Bézier curve would produce the same outcome as it made debugging slightly easier.

4.7 Reproducibility

This project was created used

- Blender `2.92` `https://www.blender.org/`
- Python `3.9.5` [6] `https://www.python.org/`
- python-chess `1.5.0` [7] `https://github.com/niklasf/python-chess`

4.7.1 Python environment

Blender is distributed with its own python installation for consistency, however this means that installed python modules are not present **blender-python-env**. To mitigate this the `--target` flag for `pip install` can be used to install directly to the blender python environment **pip-install-man**.

```
pip install -t ~/.config/blender/2.92/scripts/modules chess
```

This ensures Blenders `Python` will has access to the required libraries for this script to function.

5 Results

6 Evaluation

[6] Blender comes bundled with this version. If the system python is used instead ensure it matches the version Blender was built with and is above `3.7` for the `__future__` module. Past `3.10` the `__future__` module is no longer required.

[7] This project requires the `Outcome` class released in `1.5.0`

7 Appendix

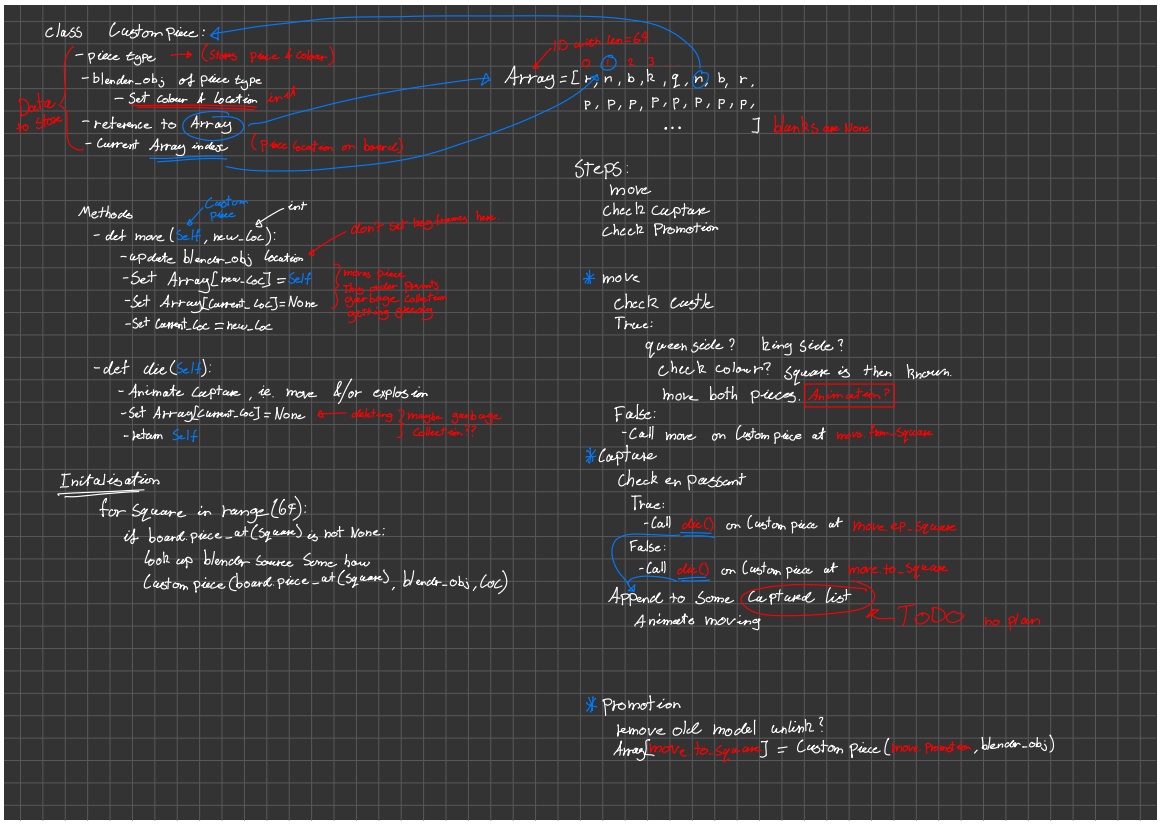


Figure 25: `CustomPiece` Initial sketch

```
class CustomPiece():
    def __init__(self, pieceType: chess.Piece, blender_obj: bpy.types.Object,\
                 array: List[Optional[CustomPiece]], loc: int):
        self._pieceType = pieceType.piece_type # int
        self._colour = pieceType.color         # bool
        self._blender_obj = blender_obj.copy()
        self._array = array                    # reference to array containing self
        self._inital_loc = loc
        self._loc = loc                        # int (1d array index)

        x, y = square_to_world_space(self._loc)
        self._blender_obj.location = Vector((x, y, 0.3))

        # set material based on colour
        if self._colour:
            self._mat = bpy.data.materials["White pieces"]
        else:
            self._mat = bpy.data.materials["Black pieces"]
        self._blender_obj.active_material = self._mat
```

```
        if self._colour and self._pieceType == chess.KNIGHT:
            self._blender_obj.rotation_euler[2] = radians(180) #XYZ
        # add object to collection so its visable
        bpy.data.collections[['Black',
        ↪  'White'][self._colour]].objects.link(self._blender_obj)

    def move(self, new_loc: int, zTo: float = 0.3):
        xTo, yTo = square_to_world_space(new_loc)
        self._blender_obj.location = Vector((xTo, yTo, zTo))
        print("Moved to ", self._blender_obj.location)

        self._array[new_loc] = self
        self._array[self._loc] = None

        self._loc = new_loc

    def die(self) -> CustomPiece:
        self._array[self._loc] = None
        self.keyframe_insert(data_path="location", frame=FRAME_COUNT-6)

        xTo, yTo = square_to_world_space(self._loc)
        self._blender_obj.location = Vector((xTo, yTo, 2.1))
        self.keyframe_insert(data_path="location", frame=FRAME_COUNT+3)

        if self._colour:
            self._inital_loc += -16
        else:
            self._inital_loc += 16

        xTo, yTo = square_to_world_space(self._inital_loc)
        self._blender_obj.location = Vector((xTo, yTo, 2.1))
        self.keyframe_insert(data_path="location", frame=FRAME_COUNT+21)

        xTo, yTo = square_to_world_space(self._inital_loc)
        self._blender_obj.location = Vector((xTo, yTo, 0.1))
        self.keyframe_insert(data_path="location", frame=FRAME_COUNT+29)

        return self
```