

Chess game animation in blender

Jake Moss - s46409665

June 6, 2021

Contents

1	Introduction	3
1.1	Project aims	3
2	The plan	3
3	Blender implementation	3
3.1	Modelling, textures, and shading	3
3.1.1	Nodes	3
3.1.2	Shaders	4
3.1.3	Pieces	5
3.1.4	Board	7
3.2	Particle effects	8
3.2.1	Explosions	8
3.2.2	Confetti	8
3.3	Lighting	8
3.3.1	Direct	8
3.3.2	Indirect	8
3.3.3	Disco ball	8
3.4	Render engine	8
3.4.1	Eevee	8
3.4.2	Cycles	8
3.4.3	Luxcore	9
3.4.4	Tragedy - 22:20, 01/June/2021	9
4	Python implementation	9
4.1	Processing games	9
4.2	Pairing problem	10
4.3	The solution	10
4.4	Array index to world space	10
4.4.1	Abuse of this functionality	12
4.5	Special moves	13
4.5.1	Castling	13
4.5.2	En passant	14
4.5.3	Promotion	14

4.6	Animation	14
4.6.1	Key frames	14
4.6.2	Interpolation	15
4.7	Reproducibility	16
4.7.1	Python environment	16
5	Results	16
6	Evaluation	16
7	Appendix	17

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 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. [1]

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 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 compute 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). [2]

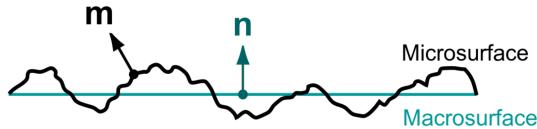


Figure 1: Micro vs macro surface (Source)

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.

- Distribution¹

- GGX

GGX is a BRDF (bidirectional reflection distribution function) which aims to be 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. [2]

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 [3].

- 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

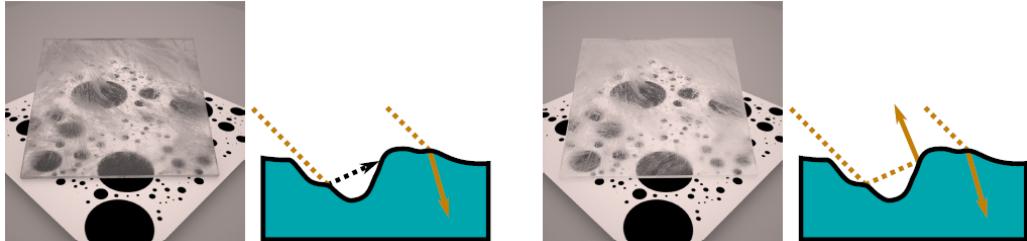


Figure 2: Single scattering (left), multiple scatters (right), (Source)

microfacets. Omitting outgoing light breaks conservation of energy and leads to dark patches within rough surfaces [4].

Blenders **Multiple-scattering GGX BRDF** allows for multiple light bounces within microfacets to achieve 100% energy conservation and provide a more physically accurate render [3], [4]. 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 [5]. 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 [5].

- 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 [6]. TODO
 - Random walk
... This comes at a cost of rendering time (actually performance hit is largely dependent on the model itself), and increased noise. TODO

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.

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 Mesh Extra Objects add-on. To transform this line of vertices to a solid object a **Screw** modifier was applied.

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

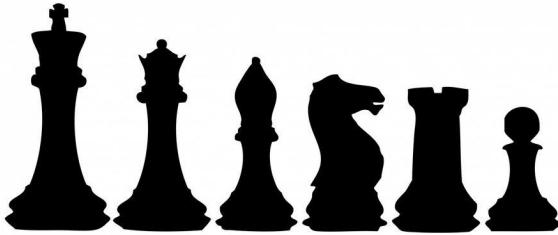
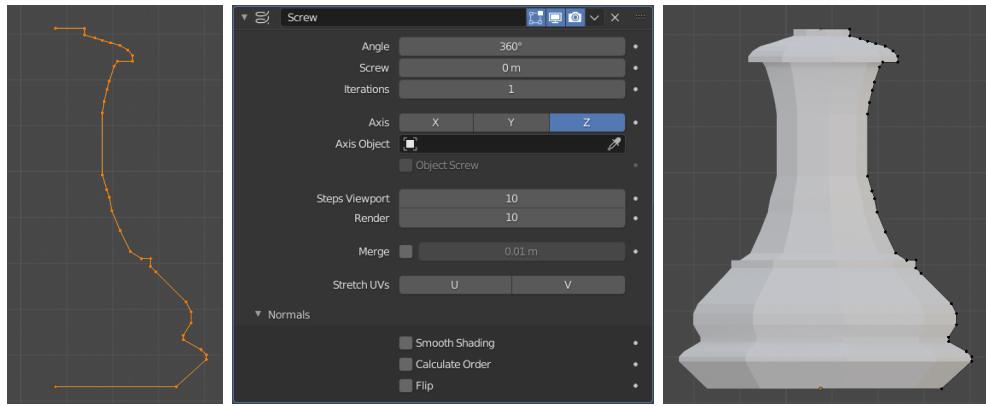


Figure 3: Reference image, Licensed under Pixabay License



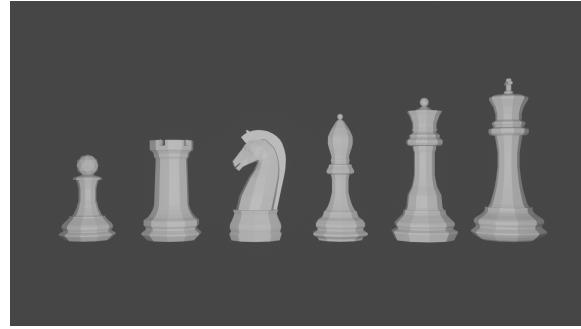
The notable changes from the default settings is the lowering of the steps from 16 → 10 and disabling **Smooth Shading**. This was a stylistic choice as it was believed that the low polygon look would better demonstrate reflections and the planned indirect lighting (See Lighting - Disco Ball).

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).



Figure 4: Knight reference images, (Source)

Additionally ico-spheres where added to piece some pieces additional detail. The finally piece models appear as below.



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.

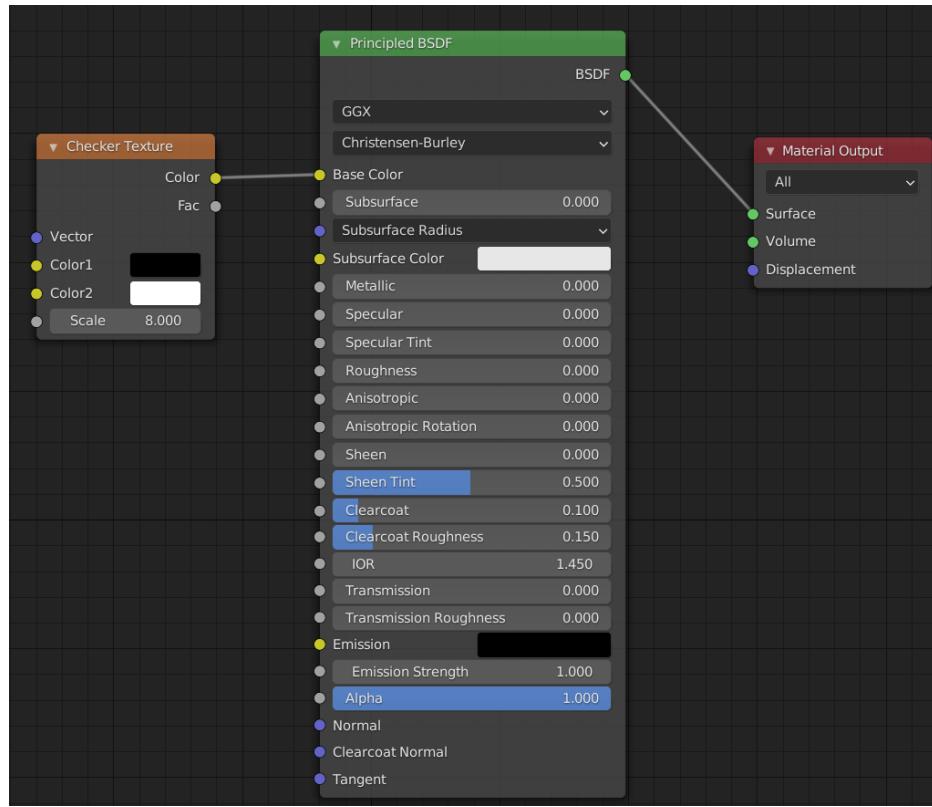


Figure 5: Complete checker board texture.

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

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 baking and rendering time costs were deemed obscene. A demo render was created and can be viewed here [TODO FIX THIS](#)

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, however the outgoing particles are set to a collection of “confetties”. The source of the explosion is a upwards facing dome, this gives the confetti a nice 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.

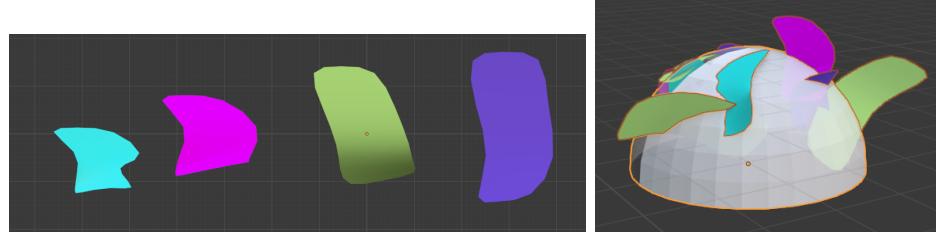


Figure 6: Confetties (left), confetti source (right)

A sample render can be seen below.

3.3 Lighting

3.3.1 Direct

3.3.2 Indirect

3.3.3 Disco ball

3.4 Render engine

3.4.1 Eevee

3.4.2 Cycles

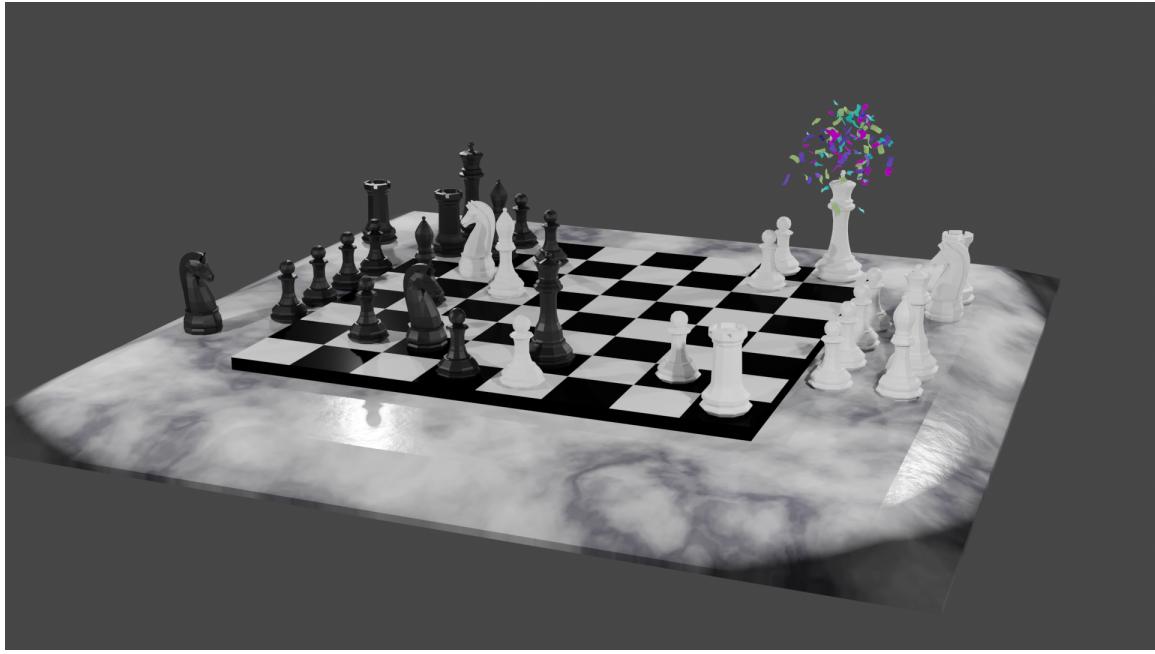


Figure 7: Confetti render using Cycles

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

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) # Pushes 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 12.

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)

```

8	56	57	58	59	60	61	62	63
7	48	49	50	51	52	53	54	55
6	40	41	42	43	44	45	46	47
5	32	33	34	35	36	37	38	39
4	24	25	26	27	28	29	30	31
3	16	17	18	19	20	21	22	23
2	8	9	10	11	12	13	14	15
1	0	1	2	3	4	5	6	7

a b c d e f g h

A	B	C	D	E	F	G	H		
8	56	57	58	59	60	61	62	63	8
7	48	49	50	51	52	53	54	55	7
6	90	91	92	93	94	95	96	97	6
5	32	33	34	35	36	37	38	39	5
4	24	25	26	27	28	29	30	31	4
3	16	17	18	19	20	21	22	23	3
2	8	9	10	11	12	13	14	15	2
1	0	1	2	3	4	5	6	7	1
	1	2	3	4	5	6	7	8	

Figure 8: Array representation ((t1) Source code, (tr) Chess board, (b) Overlaid)

To convert from array indexing two simple expressions were used.

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

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

² 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 → 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 9: Extended conversion

²Note div here is integer division.

4.5 Special moves

Figure 10 shows the main loop logic, used to move the correct pieces.

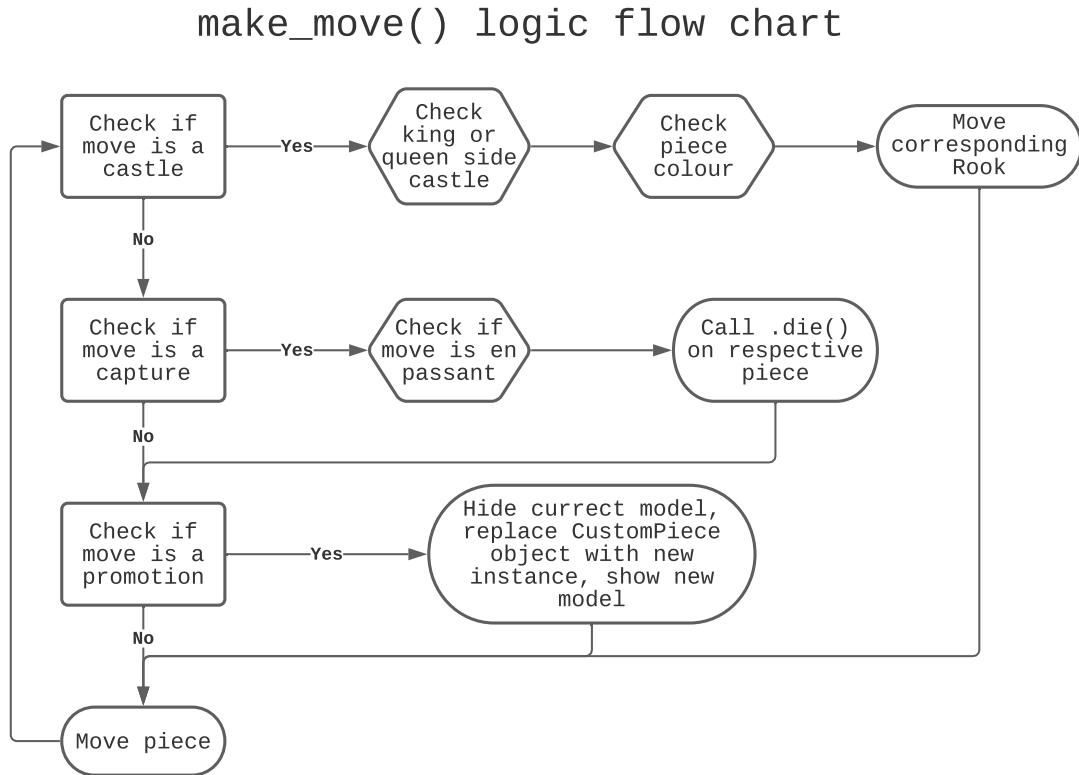


Figure 10: Main loop logic

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.

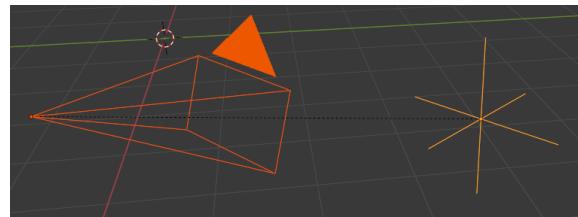


Figure 11: Camera parent axis

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 [7].

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 ³ <https://www.python.org/>
- python-chess 1.5.0 ⁴ <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 [8]. To mitigate this the `--target` flag for `pip install` can be used to install directly to the blender python environment [9].

```
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

³Blender 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.

⁴This project requires the `Outcome` class released in 1.5.0

7 Appendix

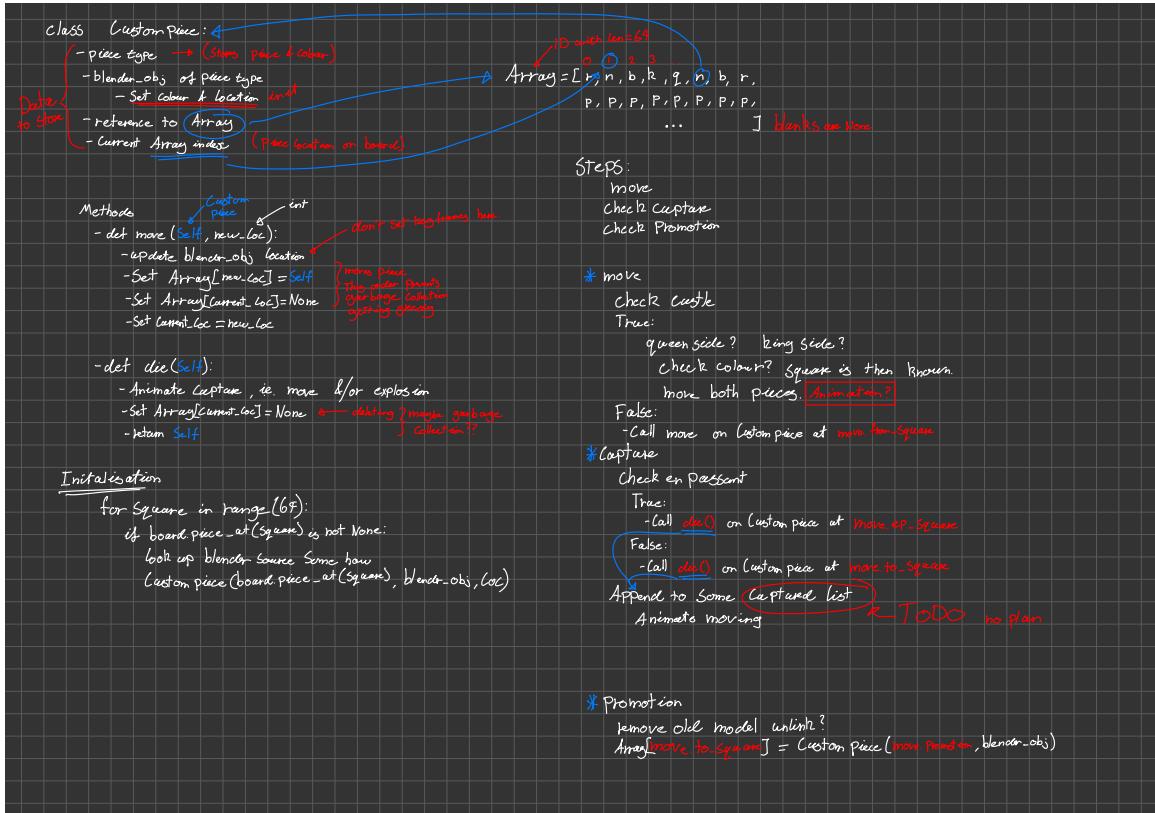


Figure 12: CustomPiece Initial sketch

```
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._initial_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._initial_loc += -16
    else:
        self._initial_loc += 16

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

    return self

```

References

- [1] H. Hedin, “Comparison of node based versus layer based compositing,” *Department of Industrial Development, IT and Land Management*, Jun. 2010. [Online]. Available: <https://www.diva-portal.org/smash/get/diva2:327273/FULLTEXT02.pdf> (cit. on p. 3).
- [2] B. Walter, S. R.Marschner, H. Li, and K. E.Torrance, “Microfacet models for refraction through rough surfaces,” 2007. [Online]. Available: <https://www.cs.cornell.edu/~srm/publications/EGSR07-btdf.pdf> (cit. on p. 4).
- [3] B. Foundation. “Principled bsdf - blender documentation.” (Feb. 25, 2020), [Online]. Available: https://docs.blender.org/manual/en/latest/render/shader_nodes/shader/principled.html (cit. on pp. 4, 5).
- [4] E. Heitz, J. Hanika, E. d’Eon, and C. Dachsbacher, “Multiple-scattering microfacet bsdfs with the smith model,” 2016. [Online]. Available: https://drive.google.com/file/d/0BzvWIdpUpRx_cFV1UkFhWXdleEU/view (cit. on p. 5).
- [5] “Cycles: Add multi-scattering, energy-conserving ggx as an option to the glossy, anisotropic and glass bsdfs.” (May 18, 2016), [Online]. Available: <https://developer.blender.org/D2002> (cit. on p. 5).
- [6] P. H. Christensen and B. Burley, “Approximate reflectance profiles for efficient subsurface scattering,” *Pixar Technical Memo*, Jun. 2015. [Online]. Available: <https://graphics.pixar.com/library/ApproxBSSRDF/> (cit. on p. 5).
- [7] M. contributors. “Blender source code, curve.c.” (), [Online]. Available: [https://developer.blender.org/diffusion/B/browse/master/source/blender/blenkernel/intern/curve.c;7c9b1065895e0a6a12555075980d7a77d1dea8c7\\$1362](https://developer.blender.org/diffusion/B/browse/master/source/blender/blenkernel/intern/curve.c;7c9b1065895e0a6a12555075980d7a77d1dea8c7$1362) (cit. on p. 16).
- [8] B. Foundation. “Tips and tricks - blender python api.” (2021), [Online]. Available: https://docs.blender.org/api/current/info_tips_and_tricks.html#bundled-python-extensions (visited on 05/30/2021) (cit. on p. 16).
- [9] pip developers, *Pip-install(1) linux man page*, 20.3, Jan. 26, 2021 (cit. on p. 16).