AVG - A Voxel Game

Exploring Dyalog APL and Game Programming

Kyle Croarkin — croarkin.kyle@gmail.com

Quick Warning

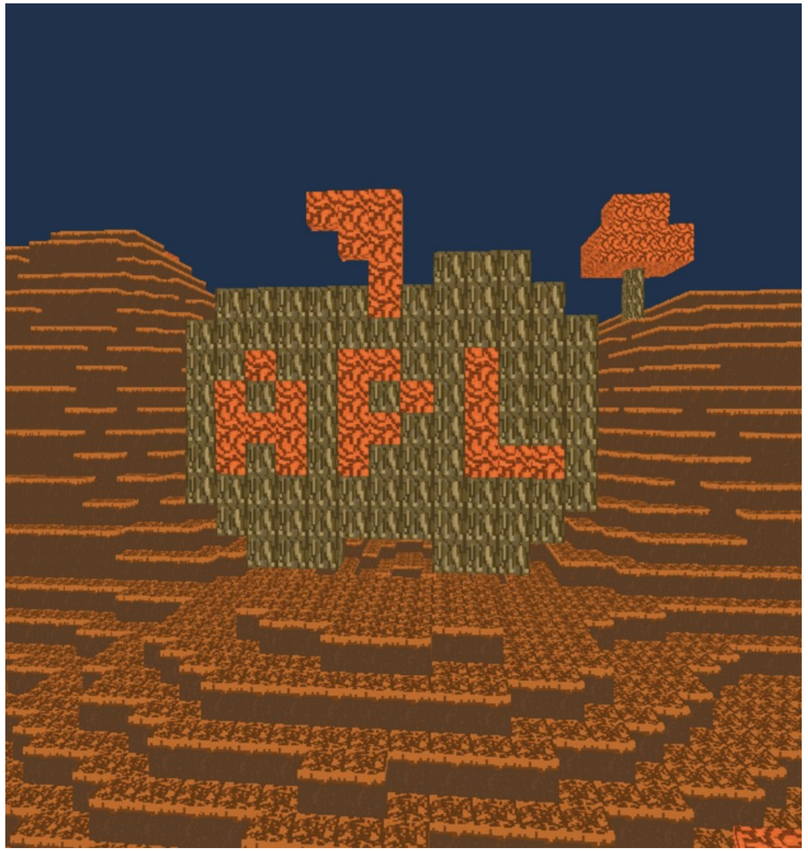
- Fair amount of APL
- Fair amount of game/graphics concepts
- Don’t worry about understanding both — enjoy the ride.

Overview

1. Overview
2. Demo
3. Code example
4. Positives
5. Difficulties
6. Conclusion

Using APL for a larger project?

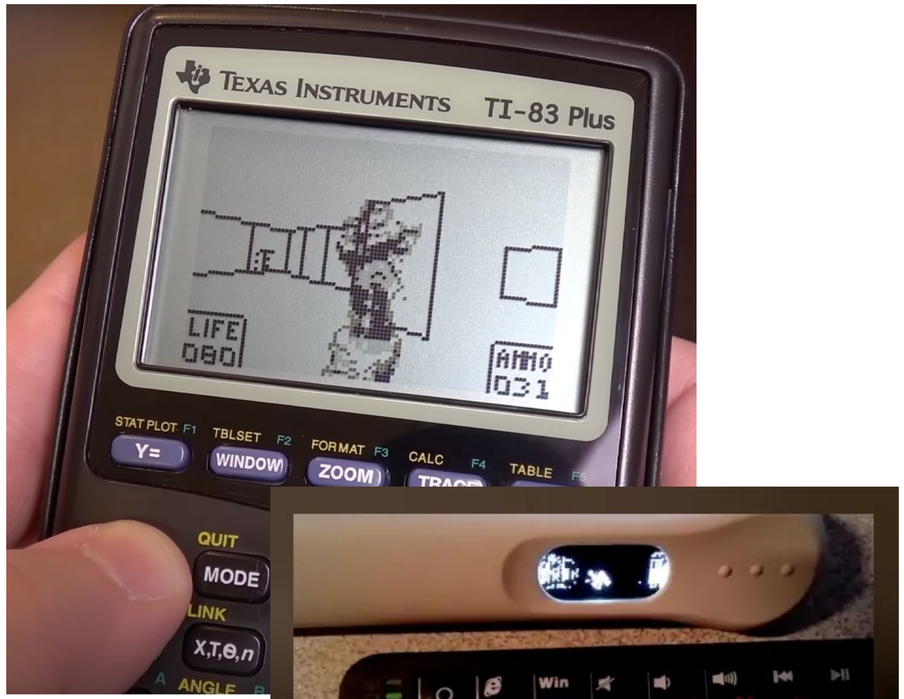
- Got fascinated by APL
- APLers make interesting claims
- Most examples online:
  - Short code snippets/Problem solving competition
  - High barrier of entry (Co-dfns/YAML parser)
- **Want to experiment APL in a domain I'm familiar with**



What to do?

People who got into programming cause of video games usually..

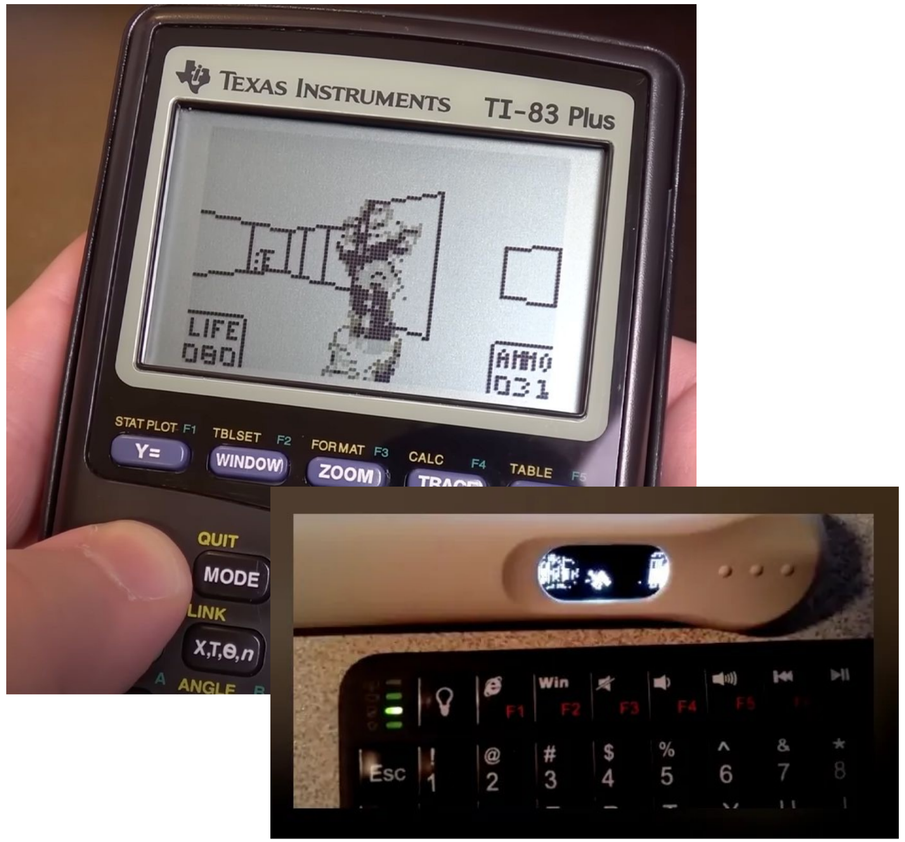
Make Doom run on it



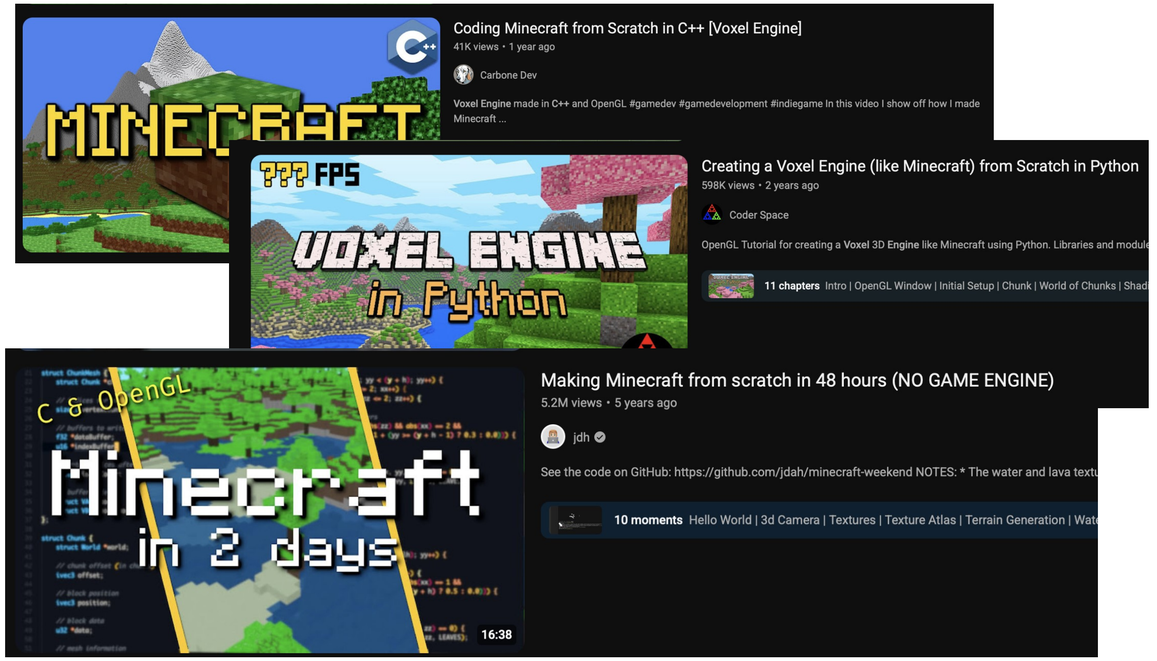
What to do?

People who got into programming cause of video games usually..

Make Doom run on it



Write a Voxel game on it



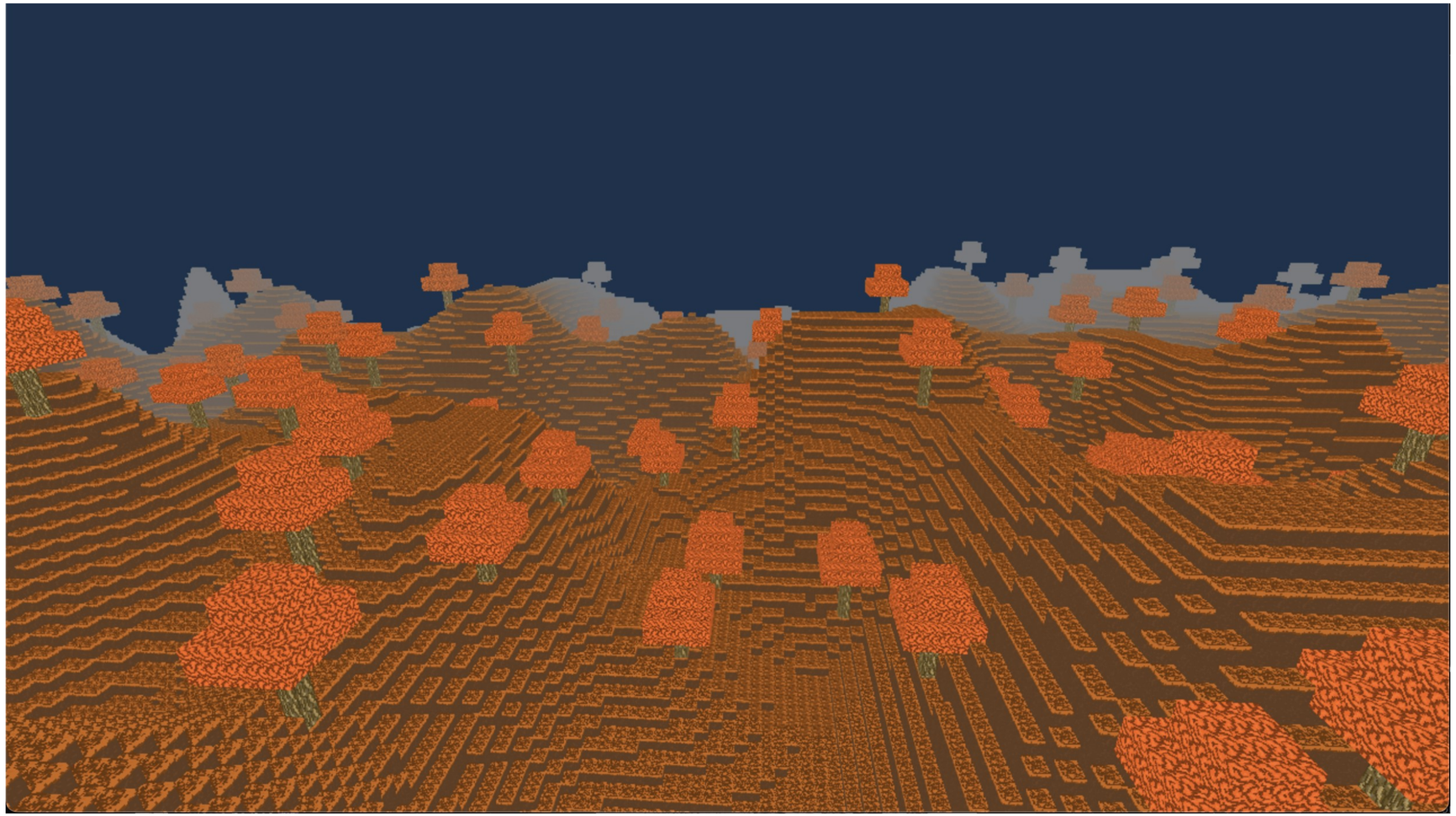
A Voxel Game?

- Famous Example: Minecraft
- World of 3d blocks that can be placed or destroyed
  - Blocks subdivided into chunks

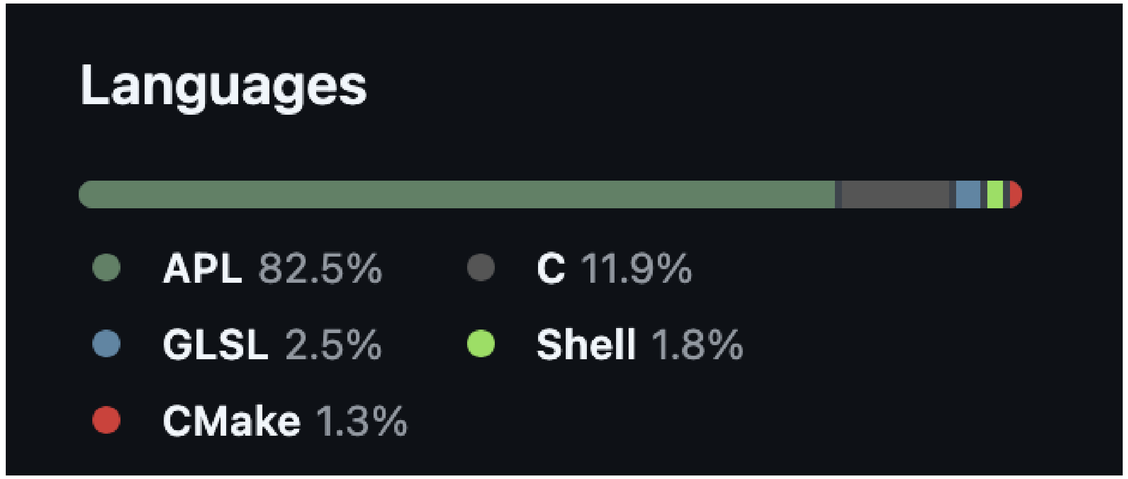
  - Need to see if language is expressive enough to handle game programming
    - And if I can handle writing APL…

Showcase

Well..?



Broad Project Structure

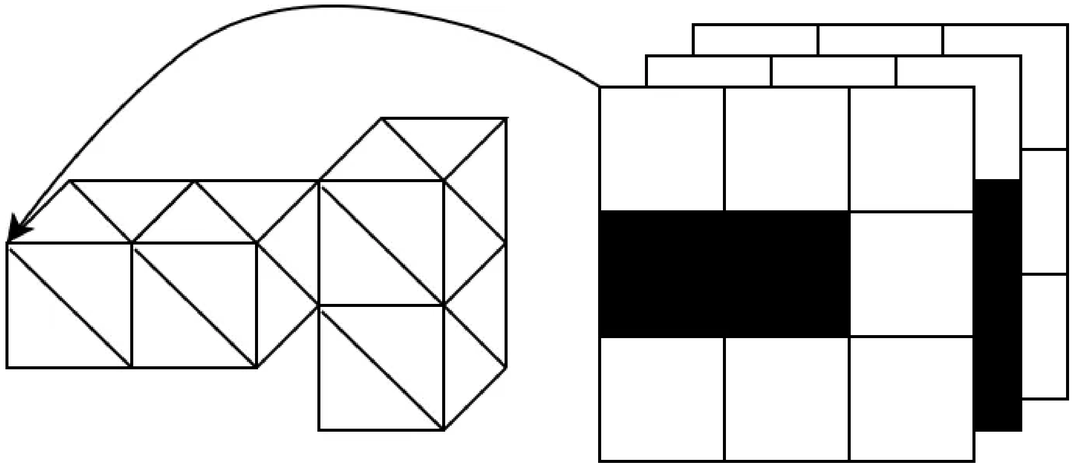


- Cross-platform library for windowing and event handling
- New low level wrapper over modern graphics apis
- Other goodies

Snippet Example

Converting chunk data to vertex data

Need to convert a 3d array of block types to geometry to be rendered.



```
l ← 16 128 16
box ← ⊃∘.∨/1@0¨l-⍥↑¨1
faces ← [
0 1 0 ◇ 0 1 1 ◇ 0 0 1 ◇ 0 0 0 ⍝ Left
1 1 1 ◇ 1 1 0 ◇ 1 0 0 ◇ 1 0 1 ⍝ Right
1 0 0 ◇ 0 0 0 ◇ 0 0 1 ◇ 1 0 1 ⍝ Bottom
0 1 0 ◇ 1 1 0 ◇ 1 1 1 ◇ 0 1 1 ⍝ Top
1 1 0 ◇ 0 1 0 ◇ 0 0 0 ◇ 1 0 0 ⍝ Back
0 1 1 ◇ 1 1 1 ◇ 1 0 1 ◇ 0 0 1 ⍝ Front
]

vertices←,[⍳2]faces+⍤1⍤2 1⊢⍉⊏⍉⍳×/l
indices←,0 1 2 0 2 3∘.+¨4×⍳4÷¨≠vertices
∇ info←Copy_chunk(chunk ptr offset);flat;solid;edges;exposed;m;vert
    flat←∊chunk
    solid←chunk≠0
    edges←solid∧box
    exposed←↑[0]{edges∨solid>⍵}¨0 1 2∘.{⍵⍋[⍺]solid}¯1 1
    vert←vertices⌿¨4/m←∊exposed
...
```

Intuition

Have one data structure representing the possible amount of faces of the chunk that can be rendered and have chunks select the faces they need when converting them

Let’s go through it with a 3-by-3-by-3 chunk.

```
      ×/3 3 3
27
```

Possible number of blocks in a chunk

Intuition

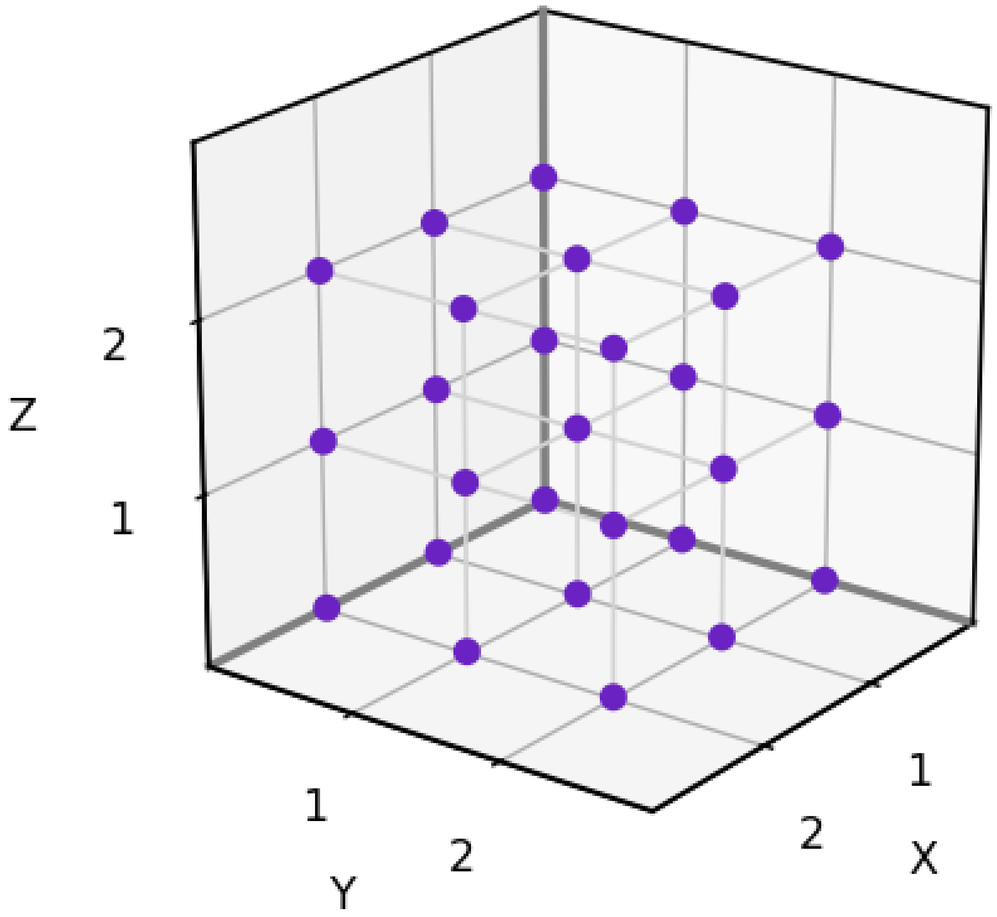
Have one data structure representing the possible amount of faces of the chunk that can be rendered and have chunks select the faces they need when converting them

Let’s go through it with a 3-by-3-by-3 chunk.

```
    ⍳×/3 3 3
0 1 2 ... 24 25 26
```

Indices of blocks in a chunk

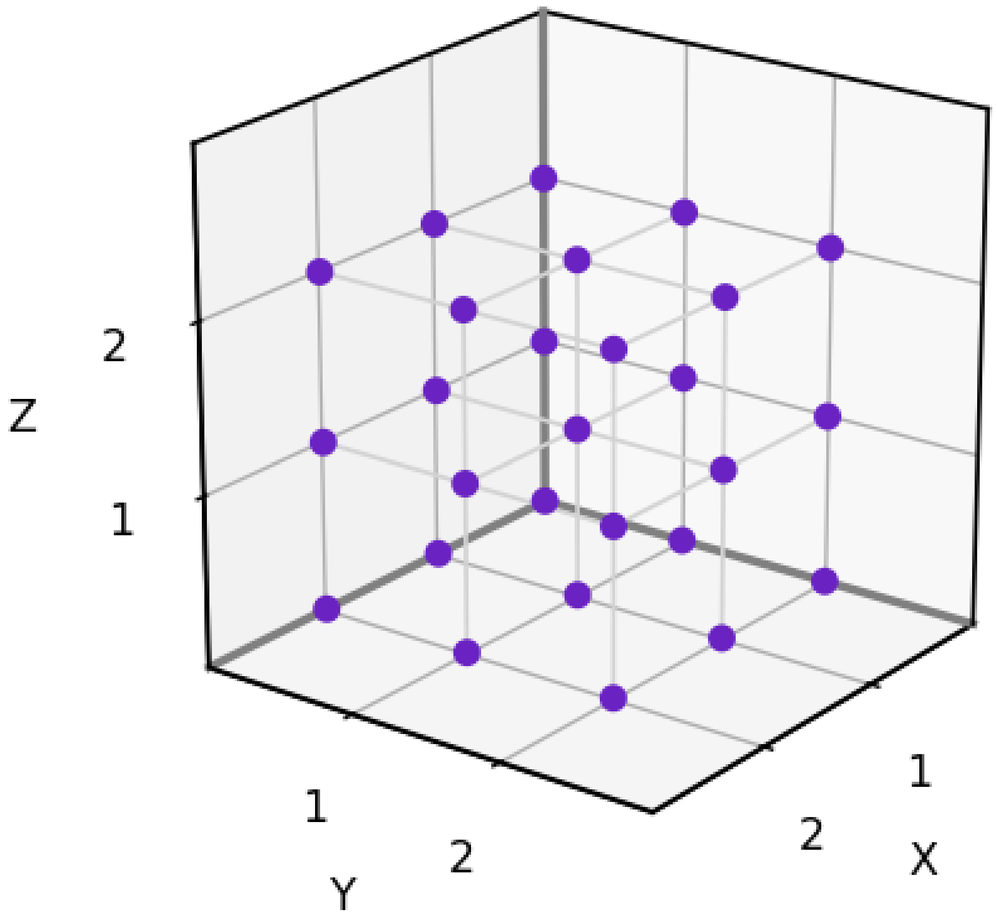
Intuition




```
      3 3 3⍴⍳×/3 3 3
0 0 0 . . . 2 2 2
0 0 0 . . . 2 2 2
0 1 2 . . . 0 1 2
```

Convert indices into “chunk size” numbering system

Intuition



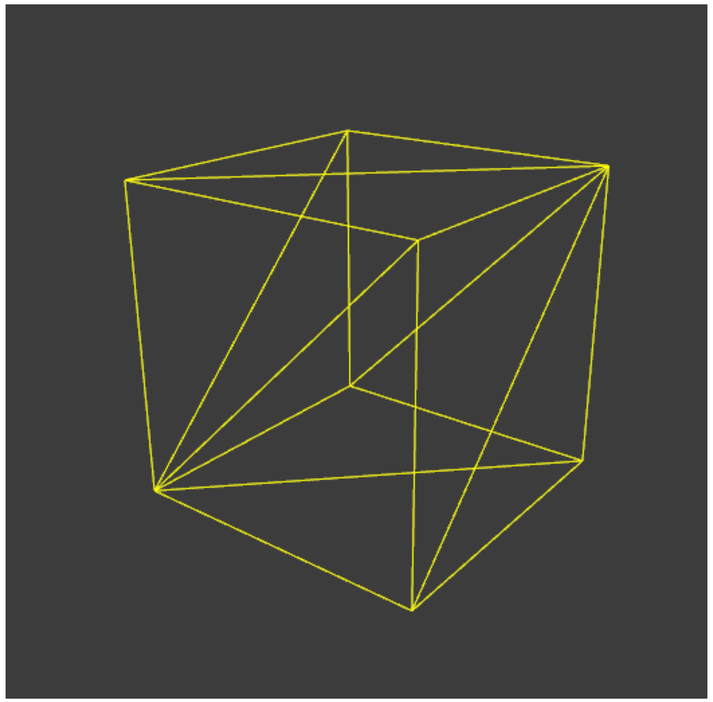

```
      ⍉3 3 3⊤⍳×/3 3 3
0 0 0
0 0 1
0 0 2

. . .

2 2 0
2 2 1
2 2 2
```

Transpose so each row is a position

Intuition



```
    faces←[
    0 1 0 ◇ 0 1 1 ◇ 0 0 1 ◇ 0 0 0 ⍝ Left
    1 1 1 ◇ 1 1 0 ◇ 1 0 0 ◇ 1 0 1 ⍝ Right
    1 0 0 ◇ 0 0 0 ◇ 0 0 1 ◇ 1 0 1 ⍝ Bottom
    0 1 0 ◇ 1 1 0 ◇ 1 1 1 ◇ 0 1 1 ⍝ Top
    1 1 0 ◇ 0 1 0 ◇ 0 0 0 ◇ 1 0 0 ⍝ Back
    0 1 1 ◇ 1 1 1 ◇ 1 0 1 ◇ 0 0 1 ⍝ Front
    ]

    l←3 3 3
    A←faces+⍤1⍤2 1⊢⍉l⍴⍳×/l
    ⍴A ⍝ Test getting shape
27 24 3
```

Add vertices of a cube to each position

Intuition

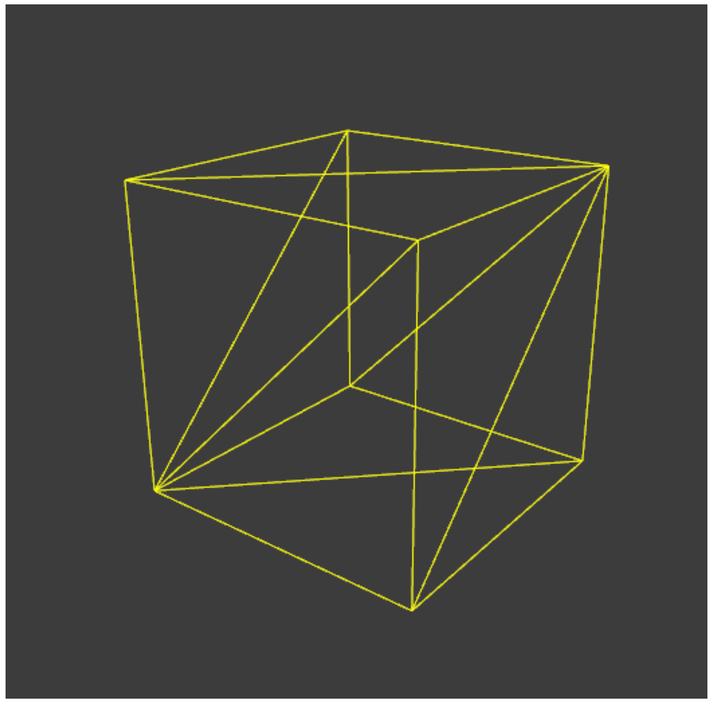
```
      ⊢A←3 2⍴⎕A
AB
CD
EF
      ⊢B←2 2⍴⎕D
01
23
      B {⎕←⍺ ⍵}⍤2 1 ⊢ A
 01  AB
 23
 01  CD
 23
 01  EF
 23
       B {⎕←⍺ ⍵}⍤1⍤2 1 ⊢ A
 01  AB
 23  AB
 01  CD
 23  CD
 01  EF
 23  EF
```

```
    faces←[
    0 1 0 ⋄ 0 1 1 ⋄ 0 0 1 ⋄ 0 0 0 ⍝ Left
    1 1 1 ⋄ 1 1 0 ⋄ 1 0 0 ⋄ 1 0 1 ⍝ Right
    1 0 0 ⋄ 0 0 0 ⋄ 0 0 1 ⋄ 1 0 1 ⍝ Bottom
    0 1 0 ⋄ 1 1 0 ⋄ 1 1 1 ⋄ 0 1 1 ⍝ Top
    1 1 0 ⋄ 0 1 0 ⋄ 0 0 0 ⋄ 1 0 0 ⍝ Back
    0 1 1 ⋄ 1 1 1 ⋄ 1 0 1 ⋄ 0 0 1 ⍝ Front
    ]

    l←3 3 3
    A←faces+⍤1⍤2 1⊢⍉⍤1 l⍴⍳×/l
    ⍴A ⍝ Test getting shape
27 24 3
```

Add vertices of a cube to each position

Intuition



```
    faces←[
    0 1 0 ◇ 0 1 1 ◇ 0 0 1 ◇ 0 0 0 ⍝ Left
    1 1 1 ◇ 1 1 0 ◇ 1 0 0 ◇ 1 0 1 ⍝ Right
    1 0 0 ◇ 0 0 0 ◇ 0 0 1 ◇ 1 0 1 ⍝ Bottom
    0 1 0 ◇ 1 1 0 ◇ 1 1 1 ◇ 0 1 1 ⍝ Top
    1 1 0 ◇ 0 1 0 ◇ 0 0 0 ◇ 1 0 0 ⍝ Back
    0 1 1 ◇ 1 1 1 ◇ 1 0 1 ◇ 0 0 1 ⍝ Front
    ]

    l←3 3 3
    A←,[⍳2]faces+⍤1⍤2 1⊢⍉l⊤⍳×/l
    ⍴A ⍝ Test getting shape
648 3
```

Collapse it in 2d form

What to do with that?

We can select relevant faces from A relatively easily by doing the following.

```
     27 ↑4 8 ⍴ Take 27 from 2, 0 rest
4 8 0 0 0 0 0 0 0 0 0 0 …
     3 3 3⍴27 ↑4 8 ⍴ Form of a 3d array
4 8 0
0 0 0
0 0 0

0 0 0
0 0 0
0 0 0

0 0 0
0 0 0
0 0 0
```

Example mock data

What to do with that?

What's the size if we flatten and duplicate each element by 24?

```
      ⍴24/27 ↑4 8
648
```

Can't we just select along rows where they're not 0?

Yeah.

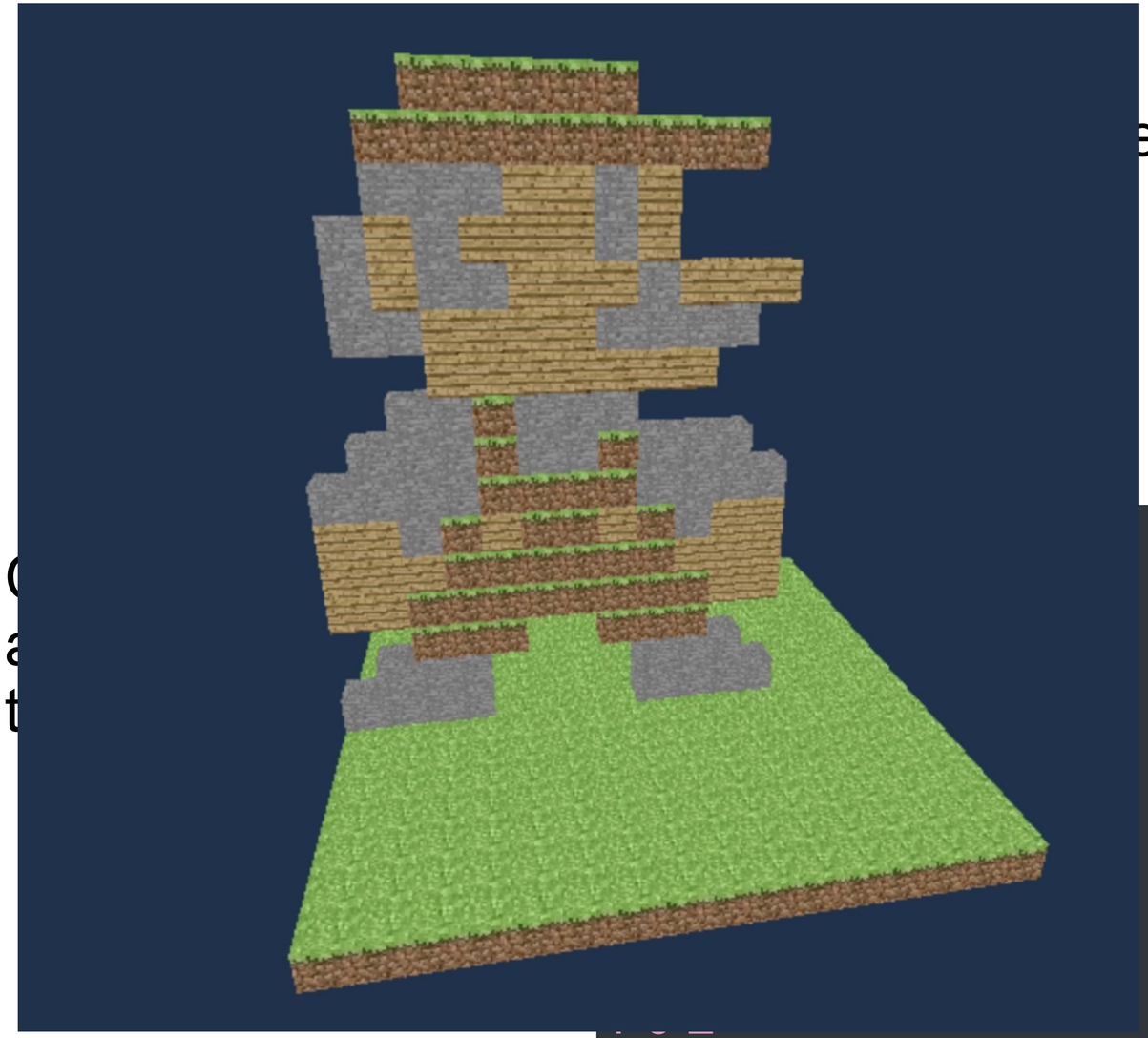
```
      (24/0≠27 ↑4 8)≠A
0 1 0
0 1 1
0 0 1
0 0 0
...
0 1 2
1 1 2
1 0 2
0 0 2
```

What if I wanted to add color or something?

No problem!

```
      B←(24/0≠27 ↑4 8)≠A
      B(,⍤1)255 0 0
0 1 0 255 0 0
0 1 1 255 0 0
0 0 1 255 0 0
0 0 0 255 0 0
...
0 1 2 255 0 0
1 1 2 255 0 0
1 0 2 255 0 0
0 0 2 255 0 0
```

What to do with that?



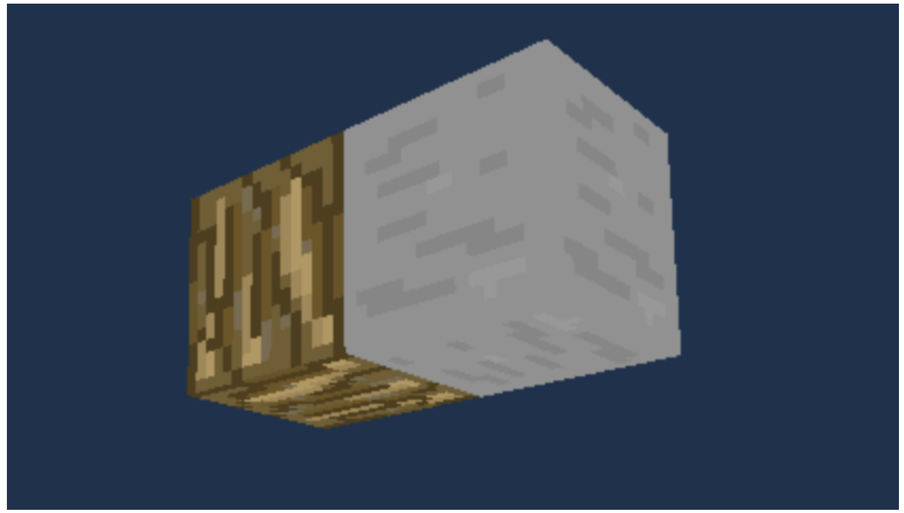
```
blocks ← world.lρ0
mario_stencil ← [
    0 0 0 1 1 1 1 1 0 0 0 0
    0 0 1 1 1 1 1 1 1 1 1 0
    0 0 2 2 2 3 3 2 3 0 0 0
    0 2 3 2 3 3 3 2 3 0 0 0
    0 2 3 2 2 3 3 3 2 3 3 3
    0 2 2 3 3 3 3 2 2 2 2 0
    0 0 0 3 3 3 3 3 3 3 0 0
          ⍝ Head Stops here
    0 0 2 2 1 2 2 2 0 0 0 0
    0 2 2 2 1 2 2 1 2 2 2 0
    2 2 2 2 1 1 1 1 2 2 2 2
    3 3 2 1 3 1 1 3 1 2 3 3
    3 3 3 1 1 1 1 1 1 3 3 3
    3 3 1 1 1 1 1 1 1 1 3 3
    0 0 1 1 1 0 0 1 1 1 0 0
    0 2 2 2 0 0 0 0 2 2 2 0
    2 2 2 2 0 0 0 0 2 2 2 2
    0 0 0 0 0 0 0 0 0 0 0 0
]


blocks[8;;] ← 17 16↑⊖mario_stencil
blocks[;1;] ← 1
```

This renders occluded faces!

True!

Rotate the chunk in each of the 6 directions and **compare it with itself** to see if the face is visible (ignoring edges)



```
[1] solid←chunk≠0
[2] edges←solid∧box
[3] exposed←↑[0]{edges∨solid>ω}¨0 1 2∘.{ω⌽[α]solid}¯1 1
[4] vert←vertices≠⍤¨4/m←∈exposed
```

```
[1] solid←chunk≠0
[2] edges←solid∧box
[3] exposed←↑[0]{edges∨solid>ω}¨0 1 2∘.{ωϕ[α]solid}⍣¯1 1
[4] vert←vertices≠⍨¨4/m←∈exposed
```

Does this look familiar?

```
life ← {⊃1 ω ∨.∧ 3 4 = +/ +⌿ ¯1 0 1 ∘.⊖ ¯1 0 1 ϕ¨ ⊂ω}
```

Broad pattern: Move around in the directions and perform computation on movements

Good example of ***Suggestivity***

“A notation will be said to be suggestive if the forms of the expressions arising in one set of problems suggest related expressions which find application in other problems” (Notation as a Tool of Thought).

Major Positives

- **APL puts data in your face**
- APL is ridiculously fast to iterate on
- APL is malleable

Entire data structure for world data and some helper functions

```
chunks ← 0ρ⍨0,⍳
cnames ← 'cx' 'cz' 'vb' 'vb_kind' 'idx_cnt' 'ready'
chunk_info ← (≢cnames)ρ⊂⍬
Ci ← cnames⍳⍥⊂
Cg←{
    ⍵≡'xz':↓⍉↑chunk_info[]⍨⊂Ci'cx' 'cz'
    ⍵≡'enc_xz':(,⍨2*16)⊥⍉⊃,/⍪⍨chunk_info[]⍨⊂Ci'cx'
'cz'
    chunk_info⊃⍨Ci ⍵
}
```

Unloading chunks out of memory

```
range←(ax az ◇ )+,⍤.,⍨(⍳1+2×view_distance)-view_distance
old←range~⍨Cg'xz'
chunks≠⍨←m←~old∊⍨Cg'xz'
set_m←(0≠Cg'vb')∧~m
vb_fenced⍪←¯1,'vb_kind'{⍉↑α ⍵}⌸{⍵/⍨set_m}⌸Cg'vb'
chunk_info←m⍤/⍨chunk_info
```

Major Positives

- **APL puts data in your face**
- APL is ridiculously fast to iterate on
- APL is malleable

With saving to disk :)

```
range←(⍺x ⍺z ⋄ )+,∘.,⍨(⍳1+2×view_distance)-
view_distance
old←range~⍨Cg'xz'
m←~old∊⍨Cg'xz'
{ ⍝ (major cell index ⋄ index into chunk hash array)
    (packR chunks[]⍨⊃⍵)⌷FREPLACE(map ⋄ 2+1⊃⍵)
}{⍺⍺¨∘↓⍨⍋(~0∊⍴⍵)⊢⍵}(~m)≠⍉[⍳≢chunks ⋄ mapi⍳Cg'enc_xz']
chunks≠⍨←m
set_m←(0≠Cg'vb')∧~m ⍝ unload only those with vbs on
GPU
vb_fenced,←¯1,'vb_kind'{⍉↑⍺ ⍵}⌸{⍵/⍨set_m}⌸Cg'vb'
chunk_info←m⌿⍨chunk_info
```

Major Positives

- **APL puts data in your face**
- APL is ridiculously fast to iterate on
- APL is malleable

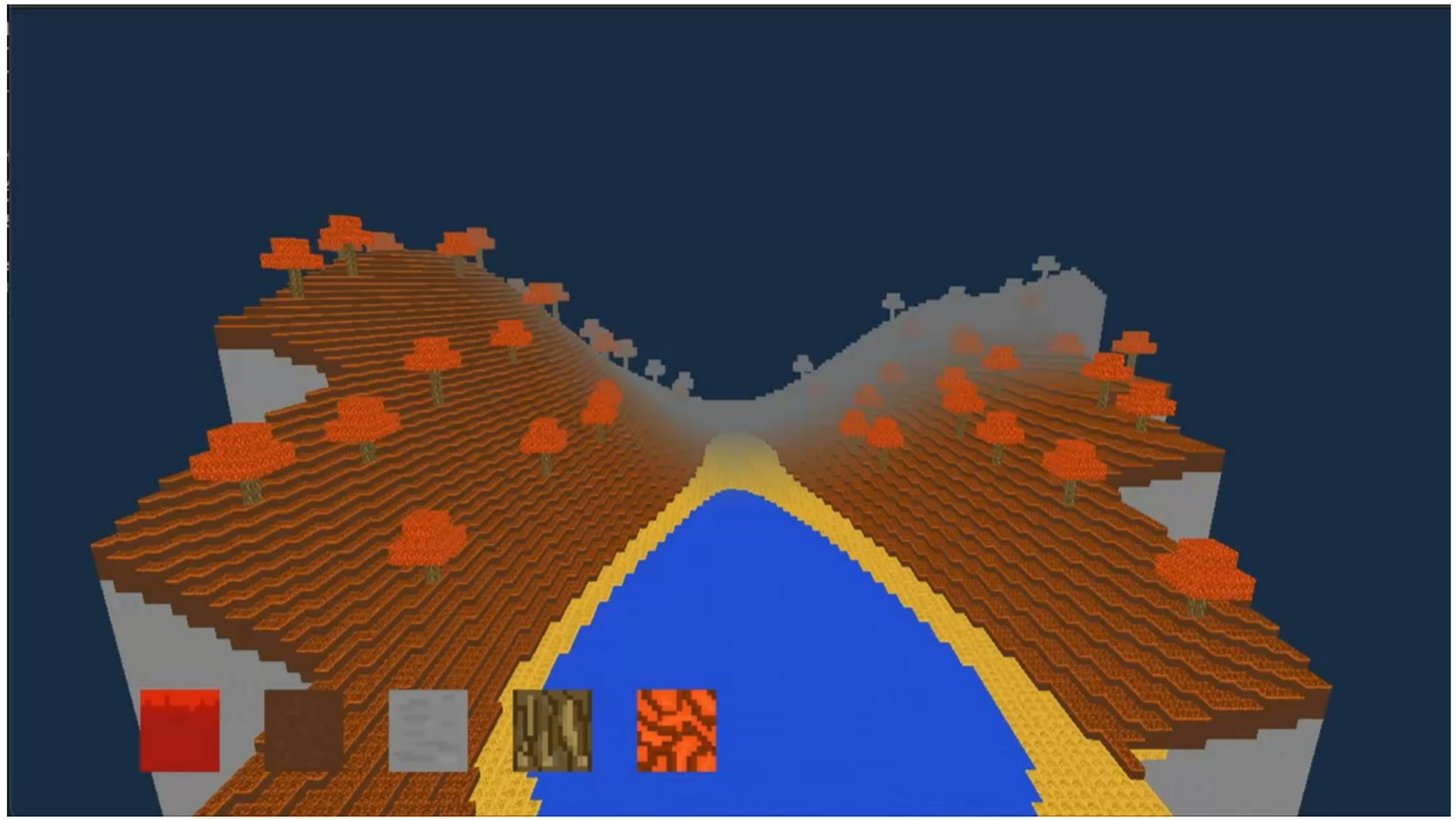
Everything world related fits into a ~500 line of code namespace file!!!

- Low level rendering and GPU buffer management
- Frustum culling
- Saving and loading world to disk
- Terrain generation with perlin noise

No need to abstract all the components when they're ***right there***.

Major Positives

- APL puts data in your face
- **APL is ridiculously fast to iterate on**
- APL is malleable



Major Positives

- APL puts data in your face
- **APL is ridiculously fast to iterate on**
- APL is malleable

```
vec←[0 0 ⋄ w 0 ⋄ 0 w ⋄ w w](+⍤2)4/[1],[0.5]↑(⊂px pz-lx lz×w×0.25)-⍨(w←⊃l)×Cg'xz'
dist←0.5*⍨+/[2]vec*2
num←(lx lz)(+.×⍤1)↑vec
den←dist×0.5*⍨+/(lx lz)*2
m←(Cg'ready')∧∨/(○70÷180)>¯2○num(⎕DIV:1).÷den
⍝ Chance that distance could be 0
```

```
frs←##.player.Get_frustum (○70÷180⋄1280÷720⋄0.1⋄500.0)
n←3↑[1]frs
d←,3↓[1]frs
(w h)←(⊃l ⋄ 1⊃l)
box←[
    0 0 0 ⋄ w 0 0 ⋄ 0 0 w ⋄ w 0 w
    0 h 0 ⋄ w h 0 ⋄ 0 h w ⋄ w h w
]
vec←box(+⍤2)8/[1],[0.5]⌽2⌽0,(w←⊃l)×↑Cg'xz'
t←~(∧/⍤1)(∨/⍤1)0>d(-⍤1)⍨ n(+.×⍤2 1)vec
t2 ← (2×w)>({0.5*⍨+/⍵*2}⍤1)↑(⊂px pz)-⍨w×Cg'xz'
m←(t2∨t)∧(Cg'ready')
```

```
frs←##.player.Get_frustum(○70÷180 ⋄ 1280÷720 ⋄ 0.1 ⋄ 500)
n←3↑[1]frs
d←,3↓[1]frs
centers←(w÷2 ⋄ 2÷⍨h←1⊃l ⋄ w÷2)(+⍤1)⌽2⌽0,(w←⊃l)×↑Cg'xz'
r←+/(|n)(×⍤1)(w ⋄ h ⋄ w)÷2
m←(Cg'ready')∧∧/(-r)(≤⍤1)d(-⍤1)⍨n(+.×⍤2 1)centers
```

Major Positives

- APL puts data in your face
- **APL is ridiculously fast to iterate on**
- APL is malleable

```
vec←[0 0 ◇ w 0 ◇ 0 w ◇ w w](+⍤2)4/[1],[0.5]↑(⊂px pz-lx lz×w×0.25)-⍨(w←⊃l)×Cg'xz'
dist←0.5*⍨+/[2]vec*2
num←(lx lz)(+.×⍤1)↑vec
den←dist×0.5*⍨+/(lx lz)*2
m←(Cg'ready')∧∨/(○70÷180)>¯2○num(⎕DIV:1).÷den
⍝ Chance that distance could be 0
```

```
frs←##.player.Get_frustum (○70÷180◇1280÷720◇0.1◇500.0)
n←3↑[1]frs
d←,3↓[1]frs
(w h)←(⊃l ◇ 1⊃l)
box←[
    0 0 0 ◇ w 0 0 ◇ 0 0 w ◇ w 0 w
    0 h 0 ◇ w h 0 ◇ 0 h w ◇ w h w
]
vec←box(+⍤2)8/[1],[0.5]⍫2⍫0,(w←⊃l)×↑Cg'xz'
t←~(∧/⍤1)(∨/⍤1)0>d(-⍤1)⍨ n(+.×⍤2 1)vec
t2 ← (2×w)>({0.5*⍨+/⍵*2}⍤1)↑(⊂px pz)-⍨w×Cg'xz'
m←(t2∨t)∧(Cg'ready')
```

```
frs←##.player.Get_frustum(○70÷180 ◇ 1280÷720 ◇ 0.1 ◇ 500)
n←3↑[1]frs
d←,3↓[1]frs
centers←(w÷2 ◇ 2÷⍨h←1⊃l ◇ w÷2)(+⍤1)⍫2⍫0,(w←⊃l)×↑Cg'xz'
r←+/(|n)(×⍤1)(w ◇ h ◇ w)÷2
m←(Cg'ready')∧∧/(-r)(≤⍤1)d(-⍤1)⍨n(+.×⍤2 1)centers
```

Each took ~30mins-1hour and was usually done while I was half-asleep before going to bed

Major Positives

- APL puts data in your face
- APL is ridiculously fast to iterate on
- **APL is malleable**

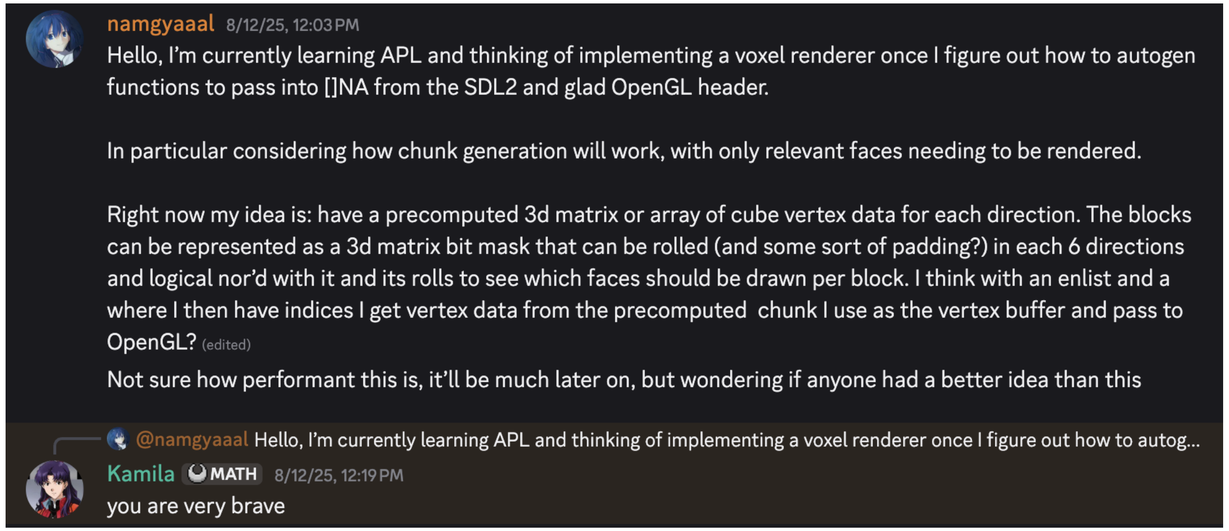
Major Positives

- APL puts data in y
- APL is ridiculously iterate on
- **APL is malleable**

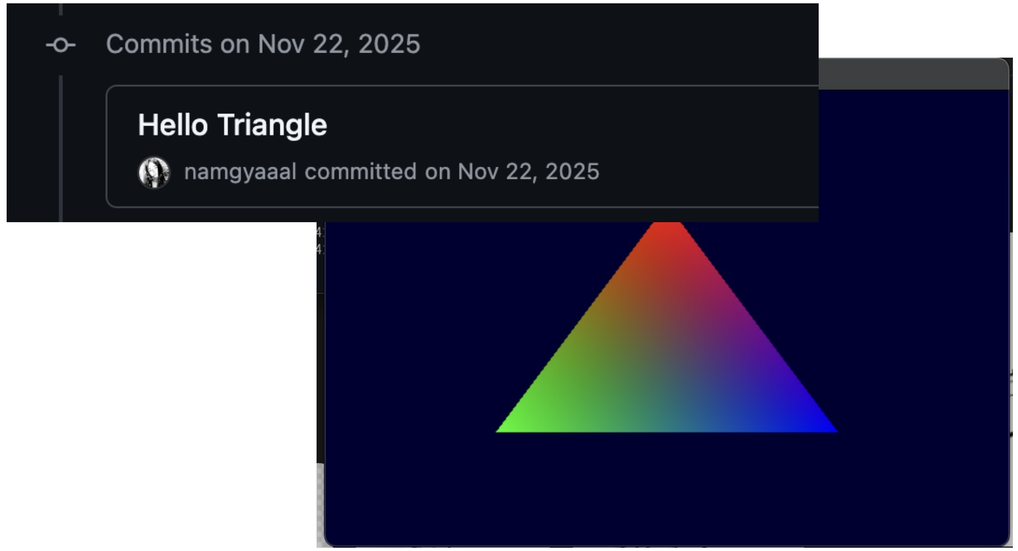
```
      ^
      ⍪2×?10⍴0
3.718573734 2.01332615 6.120604617 1.306533642 2.729847605 3.646671293 4.609359
      631 5.181575685 3.377753033 5.023003582
      ,⍪2×?10⍴0
4.041758431 0.5822016963 3.598658965 4.467057575 5.844482589 5.142105017 2.2321
      85991 4.957937923 4.444374744 1.53094216
      ⍪⍪2×?10⍴0
1.994212427
1.908670367
0.2395627673
5.472646313
2.323146086
0.6615739012
3.735301891
3.080871751
0.1106243573
0.6813307915
      1 2○⍪2×?10⍴0
RANK ERROR: Mismatched left and right argument ranks
      1 2○⍪2×?10⍴0
        ^
      1 2○⍤⍪2×?10⍴0
3.141592654 6.283185307 5.983253278 12.51191071 6.17532825 1.584038604 13.67179
      297 5.212687012 10.52519928 8.462112524 12.10651613 10.14073519
      1 2○⍤1 ⊢ ⍪2×?10⍴0
 0.9196000543   0.3928558771
 0.04307907711  0.9990716657
¯0.7276204699  ¯0.6859799209
¯0.723232587    0.6906045359
 0.9698570581   0.243674551
 0.3676708312  ¯0.9299559989
¯0.8819011065  ¯0.4714344475
 0.999997319    0.002315597508
 0.343075844    0.9393077053
 0.2067634414  ¯0.9783909644
```

Major Difficulties

- **The learning curve is a cliff**
- Domain knowledge needed
- Not all problems seem to fit well in APL



3 month time lapse…



Major Difficulties

- The learning curve is a cliff
- **Domain knowledge needed**
- Not all problems seem to fit well in APL

```
proj_mat←##.math.Ortho(0 0 600 900 0.1 100)
rot←↑{
    ul←1 ¯1 1 1×2 1 1 2○ω ⍝ Rotate on XY plane
    4 4⍴(ul,1,1)@(0 1 4 5 10 15)⊢16⍴0
}¨scope⊃⍨'rotation'
trans←↑{(⊃ω ◇ 1⊃ω ◇ 0 ◇ 1)@3⊢∘.=⍨⍳4}¨↓⍉↑x y
scale←↑{1@(⊂3 3)⊢ω×∘.=⍨⍳4}¨scope⊃⍨'scale'
mp←proj_mat+.×⍨⊃(+.×⍤2 2)/origin rot scale trans
```

```
ΔCV←{
    x←⊃⍺⊃Z ◇ a←1⊃⍺⊃Z ◇ k←⍺⊃W
    w←,[⍳3]⍉⌽[1]0 1 3 2⍉k
    Δz←ω×0<a
    ΔZ←¯2⍉¯2⌽[1](4+2↑⍴Δz)↑Δz
    Δw←⍺ Δ 3 0 1 2⍉(⍉,[⍳2]Δz)+.×,[⍳2]3 3⌹x
    Δx←w+.×⍨,[2+⍳3]3 3 SF ΔZ
}
```

Major Difficulties

- The learning curve is a cliff
- Domain knowledge needed
- **Not all problems seem to fit well in APL**

Conclusion

Definitely understand the value of APL's notation

- Want to experiment more with data structures and problems that are less amenable to APL
- Want to experiment with reading other codebases (e.g., dfns, AoC, Co-dfns)
- Made me curious on an actual full game in APL

Questions?

See outdated blogpost here:
https://homewithinnowhere.com/blog/voxel_game/

All APL code open-sourced here:
https://github.com/namgyaaal/avoxelgame