ASCII characters are not pixels: a deep dive into ASCII rendering

January 17, 2026

Recently, I've been spending my time building an image-to-ASCII renderer. Below is the result — try dragging it around, the demo is interactive!

rotating cube example.

Try opening the “split” view. Notice how well the characters follow the contour of the square.

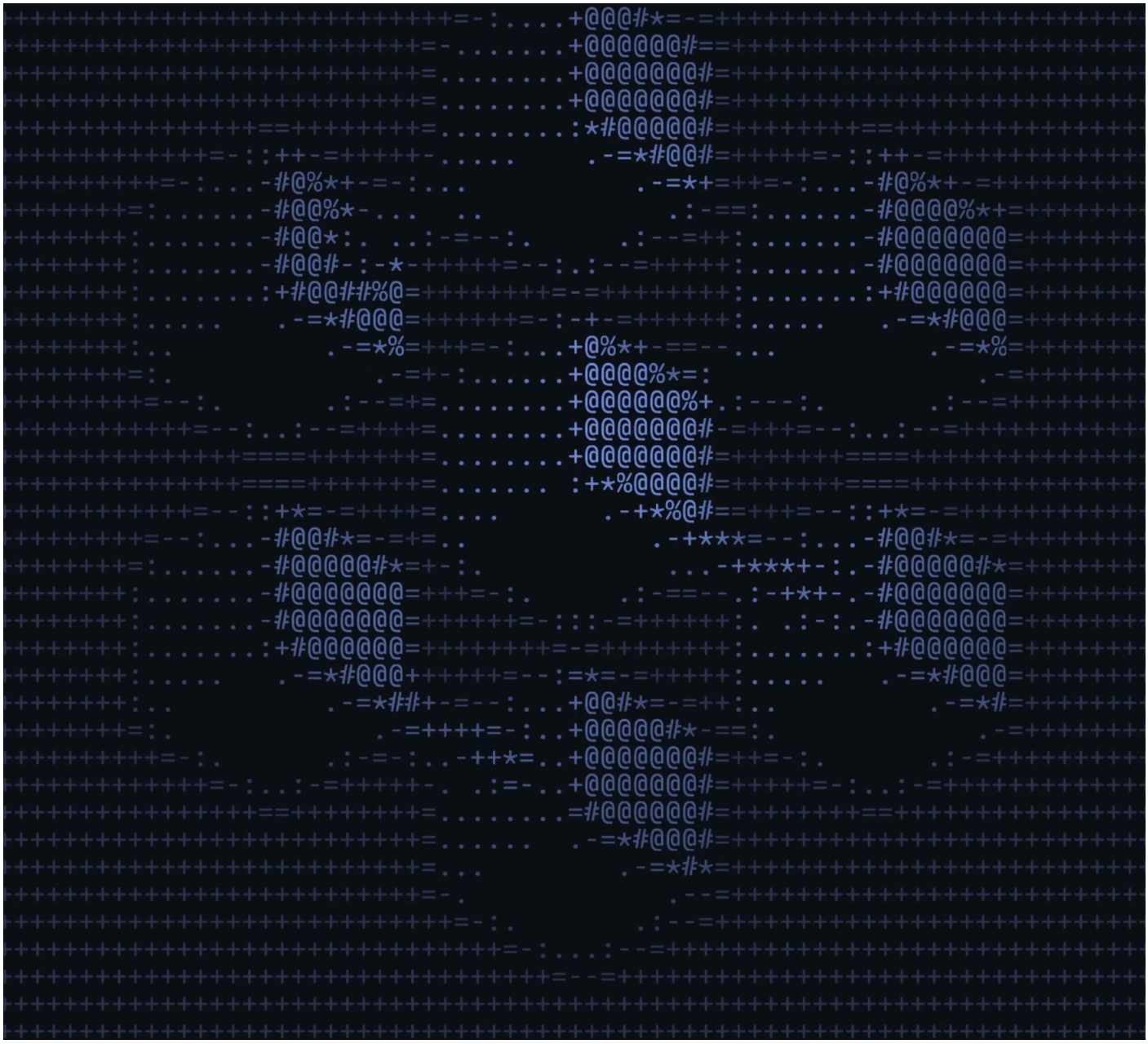
This renderer works well for animated scenes, like the ones above, but we can also use it to render static images:

The image of Saturn was generated with ChatGPT.

Then, to get better separation between different colored regions, I also implemented a cel shading-like effect to enhance contrast between edges. Try dragging the contrast slider below:

The contrast enhancement makes the separation between different colored regions far clearer. That was key to making the 3D scene above look as good as it does.

I put so much focus on sharp edges because they're an aspect of ASCII rendering that is often overlooked when programmatically rendering images as ASCII. Consider this animated 3D scene from Cognition's landing page that is rendered via ASCII characters:



Source: cognition.ai

It's a cool effect, especially while in motion, but take a look at those blurry edges! The characters follow the cube contours very poorly, and as a result, the edges look blurry and jagged in places:

This blurriness happens because the ASCII characters are being treated like pixels — their *shape* is ignored. It's disappointing to see because ASCII art looks *so much* better when shape is utilized. I don't believe I've ever seen shape utilized in generated ASCII art, and I think that's because it's not really obvious how to consider shape when building an ASCII renderer.

I started building my ASCII renderer to prove to myself that it's possible to utilize shape in ASCII rendering. In this post, I'll cover the techniques and ideas I used to capture shape and build this ASCII renderer in detail.

We'll start with the basics of image-to-ASCII conversion and see where the common issue of blurry edges comes from. After that, I'll show you the approach I used to fix that and achieve sharp, high-quality ASCII rendering. At the end, we'll improve on that by implementing the contrast enhancement effect I showed above.

Let's get to it!

Image to ASCII conversion

the following image containing a white circle using those ASCII characters:

ASCII art is (almost) always rendered using a monospace font. Since every character in a monospace font is equally wide and tall, we can split the image into a grid. Each grid cell will contain a single ASCII character.

The image with the circle is 360×360 pixels. For the ASCII grid, I'll pick a row height of 24 pixels and a column width of 20 pixels. That splits the canvas into 15 rows and 18 columns — an 18×15 grid:

Monospace characters are typically taller than they are wide, so I made each grid cell a bit taller than it is wide.

Our task is now to pick which character to place in each cell. The simplest approach is to calculate a lightness value for each cell and pick a character based on that.

We can get a lightness value for each cell by sampling the lightness of the pixel at the cell's center:

We want each pixel's lightness as a numeric value between 0 and 1, but our image data consists of pixels with RGB color values.

We can use the following formula to convert an RGB color (with component values between 0 and 255) to a lightness value:

$$\frac{R \times 0.2126 + G \times 0.7152 + B \times 0.0722}{255}$$

See relative luminance.

Now that we have a lightness value for each cell, we want to use those values to pick ASCII characters. As mentioned before, ASCII has 95 printable characters, but let's start simple with just these characters:

```
: - # = + @ * % .
```

We can sort them in approximate density order like so, with lower-density characters to the left, and high-density characters to the right:

```
. : - = + * # % @
```

We'll put these characters in a `CHARS` array:

```
const CHARS = [" ", ".", ":", "-", "=", "+", "*", "#", "%", "@"]
```

I added space as the first (least dense) character.

We can then map lightness values between 0 and 1 to one of those characters like so:

```
function getCharacterFromLightness(lightness: number) {
  const index = Math.floor(lightness * (CHARS.length - 1));
  return CHARS[index];
}
```

to high-density characters.

Rendering the circle from above with this method gives us:

That works... but the result is pretty ugly. We seem to always get `@` for cells that fall within the circle and a space for cells that fall outside.

That is happening because we've pretty much just implemented nearest-neighbor downsampling. Let's see what that means.

Nearest neighbor downsampling

Downsampling, in the context of image processing, is taking a larger image (in our case, the 360×360 image with the circle) and using that image's data to construct a lower resolution image (in our case, the 18×15 ASCII grid). The pixel values of the lower resolution image are calculated by sampling values from the higher resolution image.

The simplest and fastest method of sampling is nearest-neighbor interpolation, where, for each cell (pixel), we only take a single sample from the higher

Consider the circle example again. Using nearest-neighbor interpolation, every sample either falls inside or outside of the shape, resulting in either 0% or 100% lightness:

If, instead of picking an ASCII character for each grid cell, we color each grid cell (pixel) according to the sampled value, we get the following pixelated rendering:

These square, jagged looking edges are aliasing artifacts, commonly called jaggies. They're a common result of using nearest-neighbor interpolation.

Supersampling

To get rid of jaggies, we can collect more samples for each cell. Consider this line:

The line's slope on the y axis is $\frac{1}{3}x$. When we pixelate it with nearest-neighbor interpolation, we get the following:

Let's try to get rid of the jagginess by taking multiple samples within each cell and using the average sampled lightness value as the cell's lightness. The example below lets you vary the number of samples using the slider:

With multiple samples, cells that lie on the edge of a shape will have some of their samples fall within the shape, and some outside of it. Averaging those, we get gray in-between colors that smooth the downsampled image. Below is the same example, but with an overlay showing where the samples are taken:

This method of collecting multiple samples from the larger image is called supersampling. It's a common method of spatial anti-aliasing (avoiding jaggies at edges). Here's what the rotating square looks like with supersampling (using 8 samples for each cell):

Let's look at what supersampling does for the circle example from earlier. Try dragging the sample quality slider:

The circle becomes less jagged, but the edges feel blurry. Why's that?

Well, they feel blurry because we're pretty much just rendering a low-resolution, pixelated image of a circle. Take a look at the pixelated view:

The ASCII and pixelated views are mirror images of each other. Both are just low-resolution versions of the original high-resolution image, scaled up to the original's size — it's no wonder they both look blurry.

Increasing the number of samples is insufficient. No matter how many samples we take per cell, the samples will be averaged into a single lightness value, used to render a single pixel.

And that's the core problem: treating each grid cell as a pixel in an image. It's an obvious and simple method, but it disregards that ASCII characters have shape.

We can make our ASCII renderings far more crisp by picking characters based on their shape. Here's the circle rendered that way:

The characters follow the contour of the circle very well. By picking characters based on shape, we get a far higher *effective* resolution. The result is also more

Let's see how we can implement this.

Shape

So what do I mean by shape? Well, consider the characters `T`, `L`, and `O` placed within grid cells:

The character `T` is top-heavy. Its visual density in the upper half of the grid cell is higher than in the lower half. The opposite can be said for `L` — it's bottom-heavy. `O` is pretty much equally dense in the upper and lower halves of the cell.

We might also compare characters like `L` and `J`. The character `L` is heavier within the left half of the cell, while `J` is heavier in the right half:

We also have more “extreme” characters, such as `_` and `^`, that only occupy the lower or upper portion of the cell, respectively:

This is, roughly, what I mean by “shape” in the context of ASCII rendering. Shape refers to which regions of a cell a given character visually occupies.

Quantifying shape

To pick characters based on their shape, we’ll somehow need to quantify (put numbers to) the shape of each character.

Let’s start by only considering how much characters occupy the upper and lower regions of our cell. To do that, we’ll define two “sampling circles” for each grid cell — one placed in the upper half and one in the lower half:

It may seem odd or arbitrary to use circles instead of just splitting the cell into two rectangles, but using circles will give us more flexibility later on.

A character placed within a cell will overlap each of the cell's sampling circles to *some* extent.

One can compute that overlap by taking a bunch of samples within the circle (for example, at every pixel). The fraction of samples that land inside the character gives us the overlap as a numeric value between 0 and 1:

$$\text{Overlap} = \frac{\text{Samples inside character}}{\text{Total samples}}$$

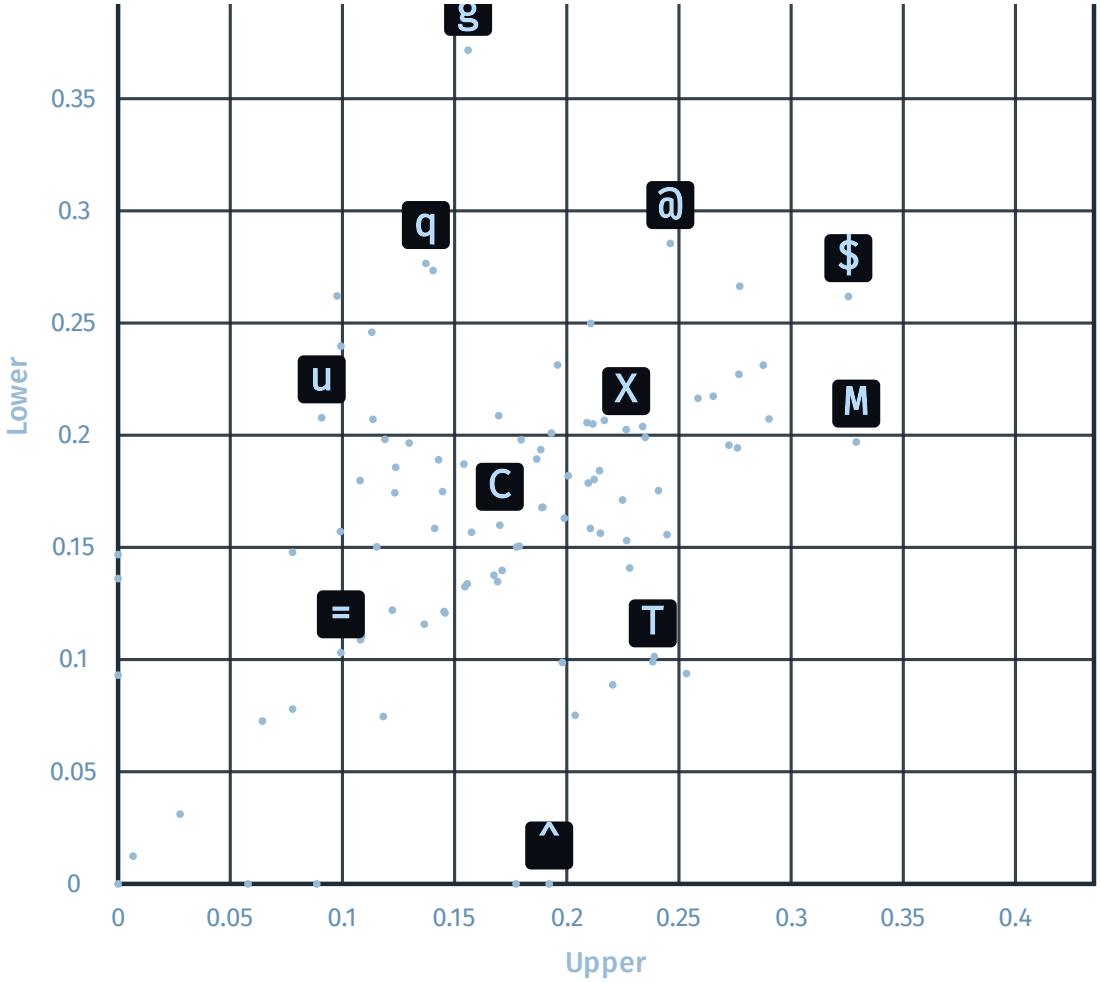
For T, we get an overlap of approximately 0.261 for the upper circle and 0.097 for the lower. Those overlap values form a 2-dimensional vector:

$$\begin{bmatrix} 0.261 \\ 0.097 \end{bmatrix}$$

We can generate such a 2-dimensional vector for each character within the ASCII alphabet. These vectors quantify the shape of each ASCII character along these 2 dimensions (upper and lower). I'll call these vectors *shape vectors*.

Below are some ASCII characters and their shape vectors. I'm coloring the sampling circles using the component values of the shape vectors:

We can use the shape vectors as 2D coordinates — here's every ASCII character on a 2D plot:



Shape-based lookup

Let's say that we have our ASCII characters and their associated shape vectors in a `CHARACTERS` array:

```
const CHARACTERS: Array<{
  character: string,
  shapeVector: number[],
}> = [ ... ];
```

We can then perform a nearest neighbor search like so:

```
function findBestCharacter(inputVector: number[]) {
  let bestCharacter = "";
```

```
  for (const { character, shapeVector } of CHARACTERS) {
    const dist = getDistance(shapeVector, inputVector);
    if (dist < bestDistance) {
      bestDistance = dist;
      bestCharacter = character;
    }
  }

  return bestCharacter;
}
```

The `findBestCharacter` function gives us the ASCII character whose shape best matches the input lookup vector.

Note: this brute force search is not very performant. This becomes a bottleneck when we start rendering thousands of ASCII characters at 60 FPS. I'll talk more about this later.

To make use of this in our ASCII renderer, we'll calculate a lookup vector for each cell in the ASCII grid and pass it to `findBestCharacter` to determine the character to display.

Let's try it out. Consider the following zoomed-in circle as an example. It is split into three grid cells:

When calculating the shape vector of each ASCII character, we took a huge number of samples. We could afford to do that because we only need to calculate those shape vectors once up front. After they're calculated, we can use them again and again.

However, if we're converting an animated image (e.g. canvas or video) to ASCII, we need to be mindful of performance when calculating the lookup vectors. An ASCII rendering might have hundreds or thousands of cells. Multiplying that by tens or hundreds of samples would be incredibly costly in terms of performance.

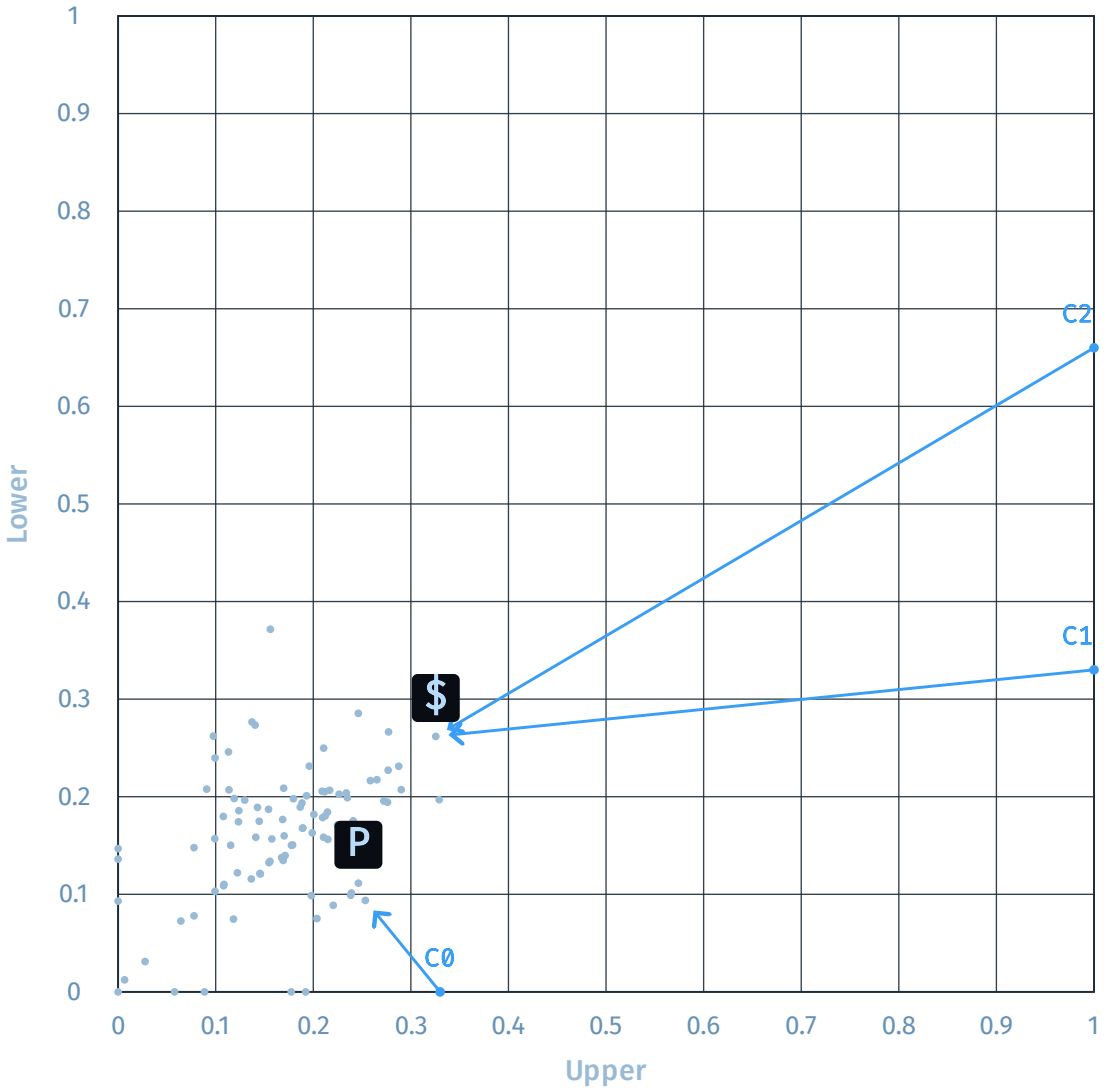
With that being said, let's pick a sampling quality of 3 with the samples placed like so:

black, giving us an average lightness of 0.33. Doing the same calculation for all of the sampling circles, we get the following 2D vectors:

$$\begin{bmatrix} 0.33 \\ 0.00 \end{bmatrix} \quad \begin{bmatrix} 1.00 \\ 0.33 \end{bmatrix} \quad \begin{bmatrix} 1.00 \\ 0.66 \end{bmatrix}$$

From now on, instead of using the term "lookup vectors", I'll call these vectors, sampled from the image that we're rendering as ASCII, *sampling vectors*. One sampling vector is calculated for each cell in the grid.

Anyway, we can use these sampling vectors to find the best-matching ASCII character. Let's see what that looks like on our 2D plot — I'll label the sampling vectors (from left to right) C0, C1, and C2:

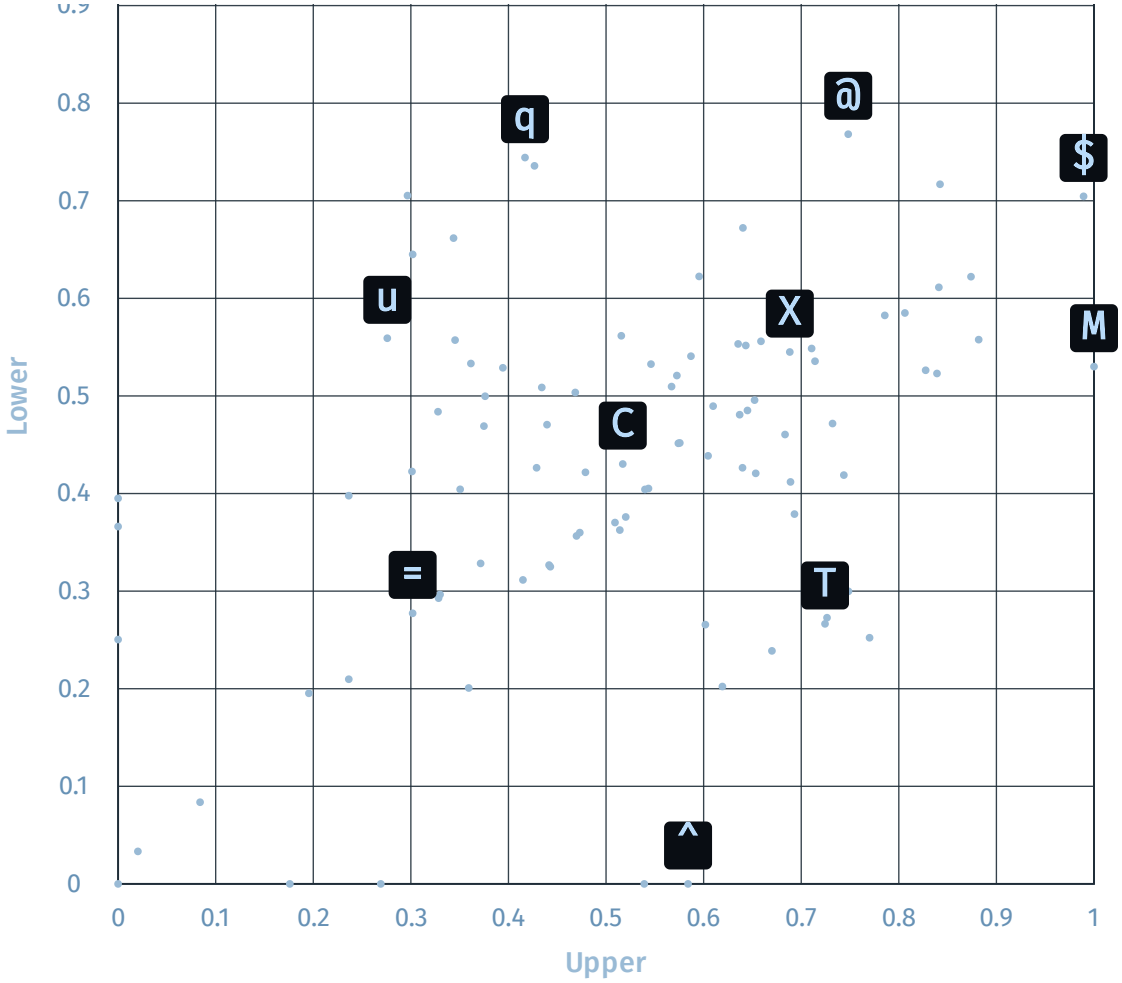


exceed 0.1, they're all clustered towards the bottom left region of our plot. This makes our sampling vectors map to a few characters on the edge of the cluster.

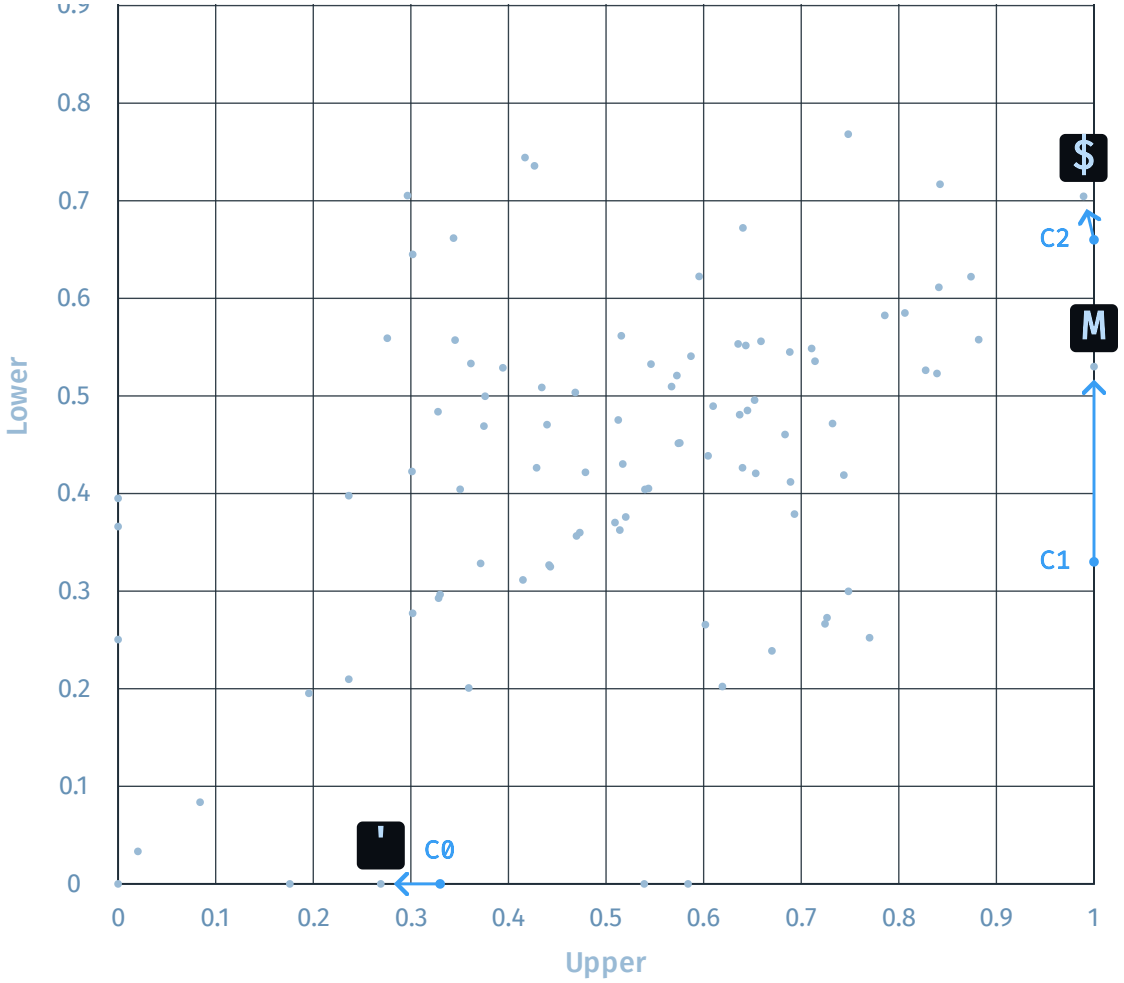
We can fix this by *normalizing* the shape vectors. We'll do that by taking the maximum value of each component across all shape vectors, and dividing the components of each shape vector by the maximum. Expressed in code, that looks like so:

```
const max = [0, 0]

for (const vector of characterVectors) {
  for (const [i, value] of Object.entries(vector)) {
    if (value > max[i]) {
      max[i] = value;
    }
  }
}

const normalizedCharacterVectors = characterVectors.map(
  vector => vector.map((value, i) => value / max[i])
)
```

Here's what the plot looks like with the shape vectors normalized:



If we now map the sampling vectors to their nearest neighbors, we get a much more sensible result:



We get `'`, `M` and `$`. Let's see how well those characters match the circle:

Nice! They match very well.

Let's try rendering the full circle from before with the same method:

Much better than before! The picked characters follow the contour of the circle very well.

Limits of a 2D shape vector

Using two sampling circles — one upper and one lower — produces a much better result than the 1-dimensional (pixelated) approach. However, it still falls short when trying to capture other aspects of a character's shape.

For example, two circles don't capture the shape of characters that fall in the middle of the cell. Consider - :

For `-`, we get a shape vector of $\begin{bmatrix} 0.029 \\ 0.002 \end{bmatrix}$. That doesn't represent the character very well at all.

The two upper-lower sampling circles also don't capture left-right differences, such as the difference between `p` and `q`:

We could use such differences to get better character picks, but our two sampling circles don't capture them. Let's add more dimensions to our shape to fix that.

Increasing to 6 dimensions

Since cells are taller than they are wide (at least with the monospace font I'm using), we can use 6 sampling circles to cover the area of each cell quite well:

6 sampling circles capture left-right differences, such as between `p` and `q`, while also capturing differences across the top, bottom, and middle regions of the cell, differentiating `^`, `-`, and `_`. They also capture the shape of "diagonal" characters like `/` to a reasonable degree.

One problem with this grid-like configuration for the sampling circles is that there are gaps. For example, `.` falls between the sampling circles:

the left sampling circles and raising the right ones) and make them a bit larger. This causes the cell to be almost fully covered while not causing excessive overlap across the sampling circles:

We can use the same procedure as before to generate character vectors using these sampling circles, this time yielding a 6-dimensional vector. Consider the character `L`:

For `L`, we get the vector:

$$\begin{bmatrix} 0.51 & 0.45 \end{bmatrix}$$

I'm presenting 6-dimensional shape vectors in a 3×2 matrix form because it's easier to grok geometrically, but the actual vector is a flat list of numbers.

The lightness values certainly look L-shaped! The 6D shape vector captures `L`'s shape very well.

Nearest neighbor lookups in a 6D space

Now we have a 6D shape vector for every ASCII character. Does that affect character lookups (how we find the best matching character)?

Earlier, in the `findBestCharacter` function, I referenced a `getDistance` function. That function returns the Euclidean distance between the input points. Given two 2D points a and b, the formula to calculate their Euclidean distance looks like so:

$$d = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}$$

This generalizes to higher dimensions:

$$d = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \cdots + (a_n - b_n)^2}$$

Put into code, this looks like so:

```
function getDistance(a: number[], b: number[]): number {
  let sum = 0;
  for (let i = 0; i < a.length; i++) {
    sum += (a[i] - b[i]) ** 2;
  }
  return Math.sqrt(sum);
}
```

So, no, the dimensionality of our shape vector does not change lookups at all. We can use the same `getDistance` function for both 2D and 6D.

With that out of the way, let's see what the 6D approach yields!

Trying out the 6D approach

Our new 6D approach works really well for flat shapes, like the circle example we've been using:

Now let's see how this approach works when we render a 3D scene with more shades of gray:

Firstly, the outer contours look nice and sharp. I also like how well the gradients across the sphere and cone look.

However, internally, the objects all kind of blend together. The edges *between* surfaces with different lightnesses aren't sharp enough. For example, the lighter faces of the cubes all kind of blend into one solid color. When there is a change in color — like when two faces of a cube meet — I'd like to see more sharpness in the ASCII rendering.

To demonstrate what I mean, consider the following split:

The different shades result in `i` s on the left and `B` s on the right, but the boundary is not very sharp.

By applying some effects to the sampling vector, we can enhance the contrast at the boundary so that it appears sharper:

The added contrast makes a *big* difference in readability for the 3D scene. Let's look at how we can implement this contrast enhancement effect.

Contrast enhancement

Consider cells overlapping a color boundary like so:

For the cells on the boundary, we get a 6D sampling vector that looks like so:

$$\begin{bmatrix} 0.65 & 0.65 \\ 0.31 & 0.31 \\ 0.22 & 0.22 \end{bmatrix}$$

To make future examples easier to visualize, I'll start drawing the sampling vector using 6 circles like so:

Currently, this sampling vector resolves to the character `T`:

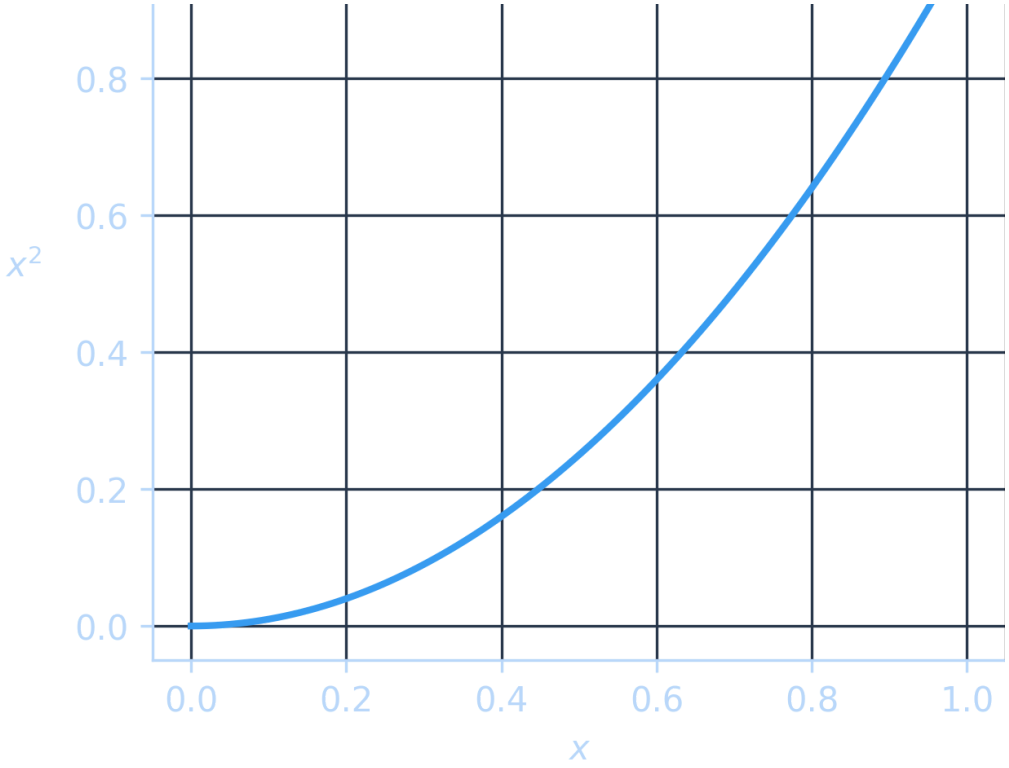
That's a sensible choice. The character `T` is visually dense in the top half and less so in the bottom half, so it matches the image fairly well.

Still, I want the picked character to emphasize the shape of the boundary better. We can achieve that by enhancing the contrast of the sampling vector.

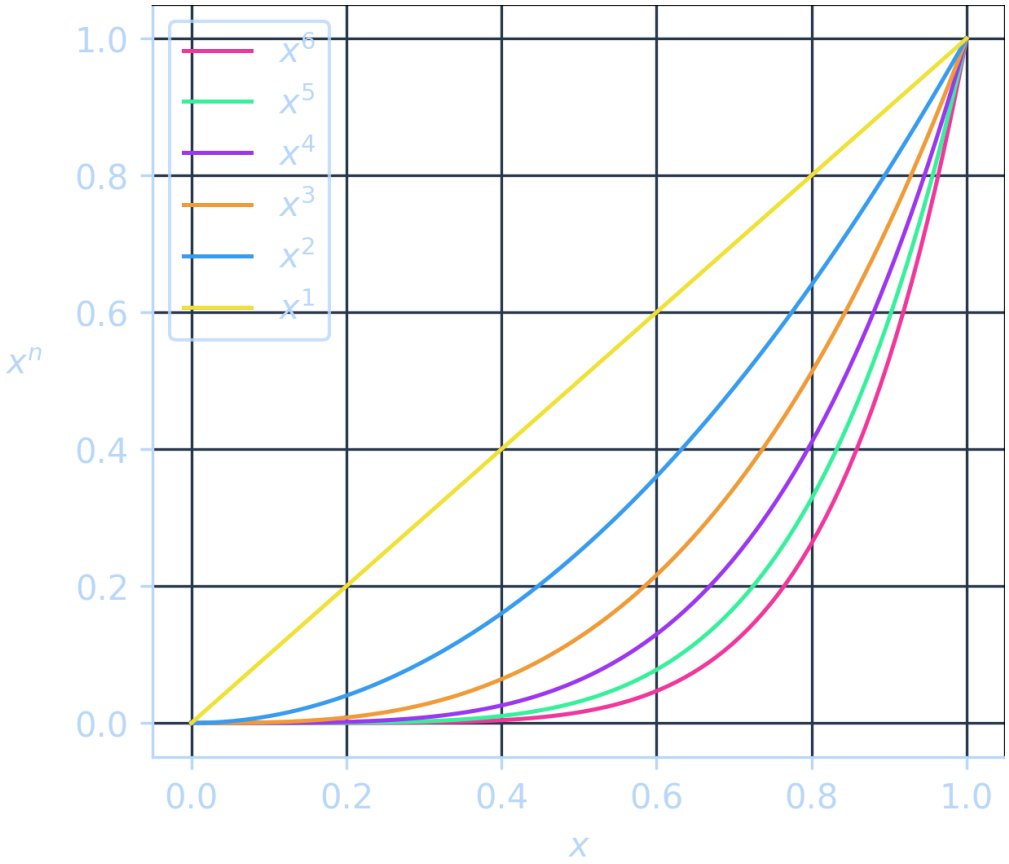
To increase the contrast of our sampling vector, we might raise each component of the vector to the power of some exponent.

Consider how an exponent affects values between 0 and 1. Numbers close to 0 experience a strong pull towards 0 while larger numbers experience less pull. For example, $0.1^2 = 0.01$, a 90% reduction, while $0.9^2 = 0.81$, only a reduction of 10%.

The level of pull depends on the exponent. Here's a chart of x^2 for values of x between 0 and 1:

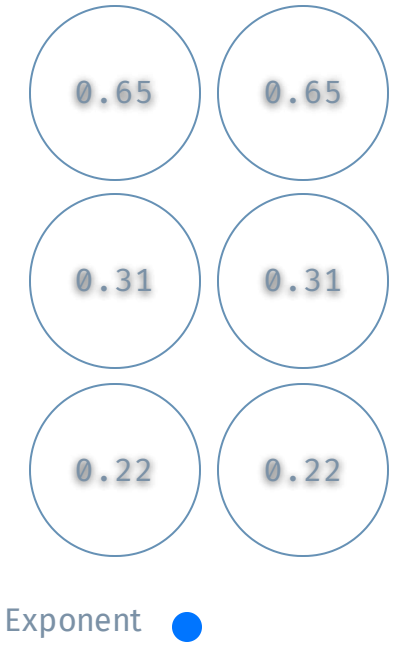


This effect becomes more pronounced with higher exponents:



A higher exponent translates to a stronger pull towards zero.

The example below allows you to vary the exponent applied to the sampling vector.



As the exponent is increased to 2, the darker components of the sampling vector quickly become *much* darker, just like we wanted. However, the lighter components also get pulled towards zero by a significant amount.

I don't want that. I want to increase the contrast *between* the lighter and darker components of the sampling vector, not the vector in its entirety.

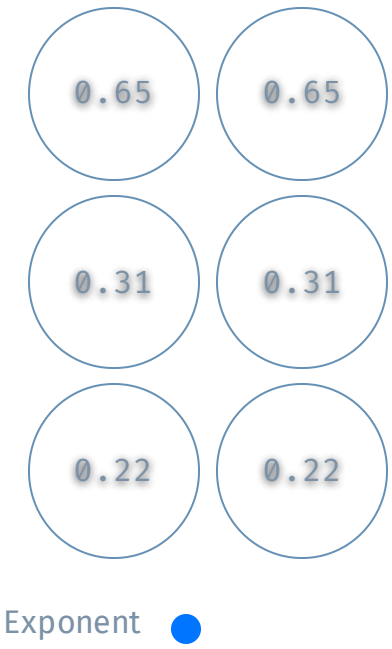
To achieve that, we can normalize the sampling vector to the range $[0, 1]$ prior to applying the exponent, and then "denormalize" the vector back to the original range afterwards.

The normalization to $[0, 1]$ can be done by dividing each component by the maximum component value. After applying the exponent, mapping back to the original range is done by multiplying each component by the same max value:

```
const maxValue = Math.max( ... samplingVector)

samplingVector = samplingVector.map((value) ⇒ {
```

```
  value = x * maxValue; // Denormalize
  return value;
})
```

Here's the same example, but with this normalization applied:



Very nice! The lightest component values are retained, and the contrast between the lighter and darker components is increased by "crunching" the lower values.

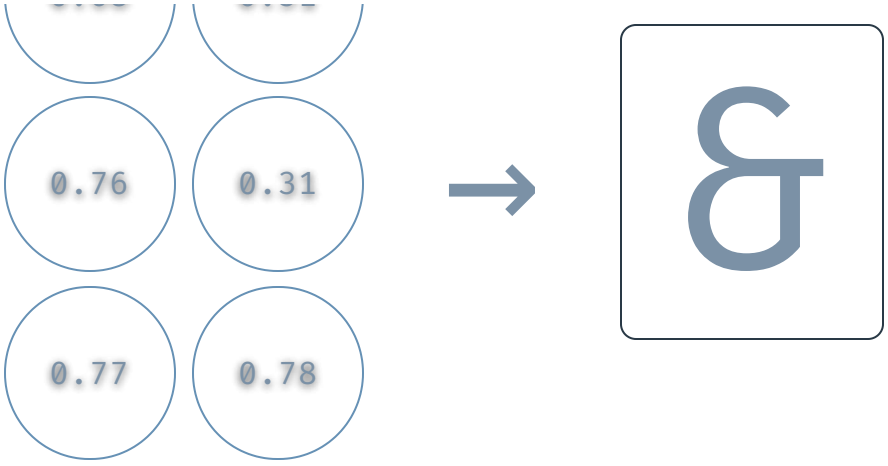
This affects which character is picked. The following example shows how the selected character changes as the contrast is increased:

Exponent

Awesome! The pick of `"` over `T` emphasizes the separation between the lighter region above and the darker region below!

By enhancing the contrast of the sampling vector, we exaggerate its shape. This gives us a character that less faithfully represents the underlying image, but improves readability as a whole by enhancing the separation between different colored regions.

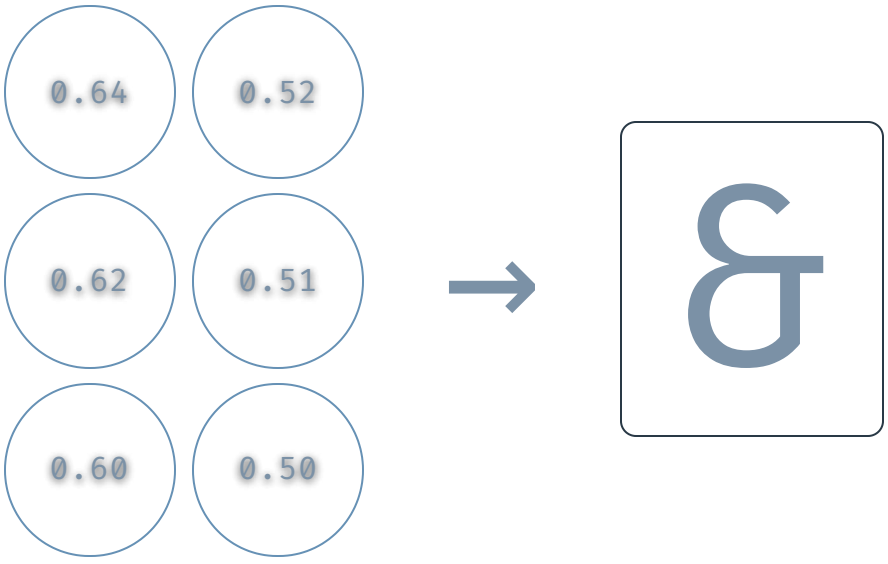
Let's look at another example. Observe how the L-shape of the sampling vector below becomes more pronounced as the exponent increases, and how that affects the picked character:



Exponent

Works really nicely! I *love* the transition from `&` → `b` → `L` as the L-shape of the vector becomes clearer.

What's nice about applying exponents to normalized sampling vectors is that it barely affects vectors that are uniform in value. If all component values are similar, applying an exponent has a minimal effect:



Exponent

Because the vector is fairly uniform, the exponent only has a slight effect and doesn't change the picked character.

We very much do *not* want to introduce unnecessary choppiness.

Compare the 3D scene ASCII rendering with and without this contrast enhancement:

We do see more contrast at boundaries, but this is not quite there yet. Some edges are still not sharp enough, and we also observe a “staircasing” effect happening at some boundaries.

Let’s look at the staircasing effect first. We can reproduce it with a boundary like so:

Below is the ASCII rendering of that boundary. Notice how the lower edge (the `!`s) becomes "staircase-y" as you increase the exponent:

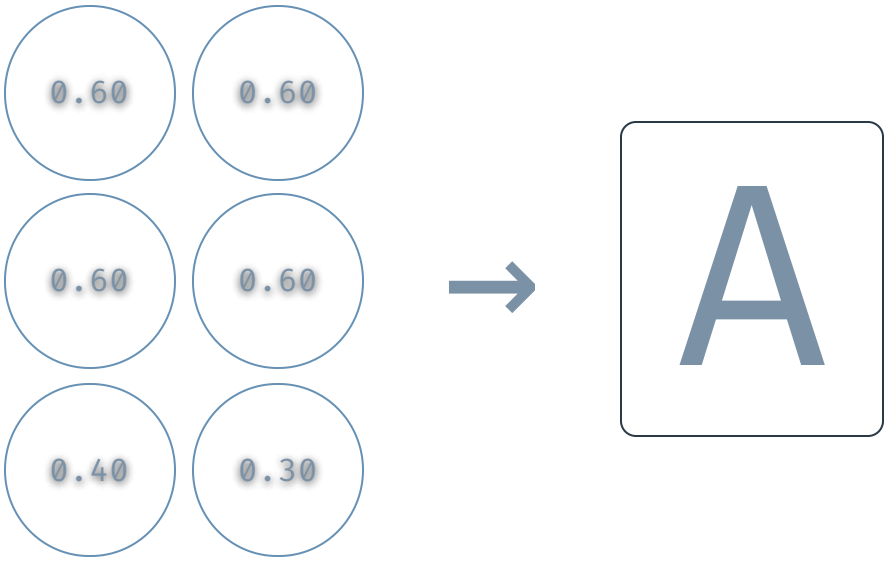
We see a staircase pattern like so:

```
                !!!!!
          !!!!!!!!!!
     !!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!
```

To understand why that's happening, let's consider the row in the middle of the canvas, progressing from left to right. As we start off, every sample is equally light, giving us `U`s:

```
UUUUUUUU →
```

darker components are enhanced by contrast enhancement, giving us some `f`s:

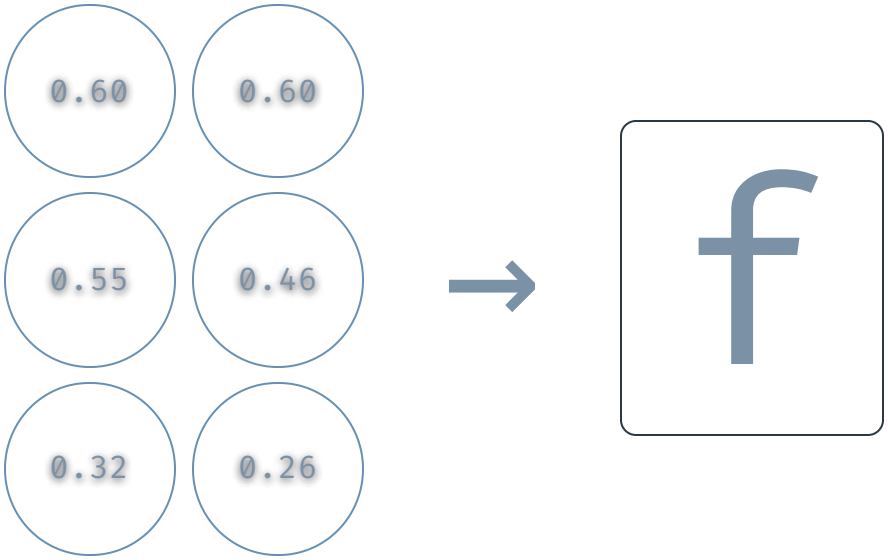


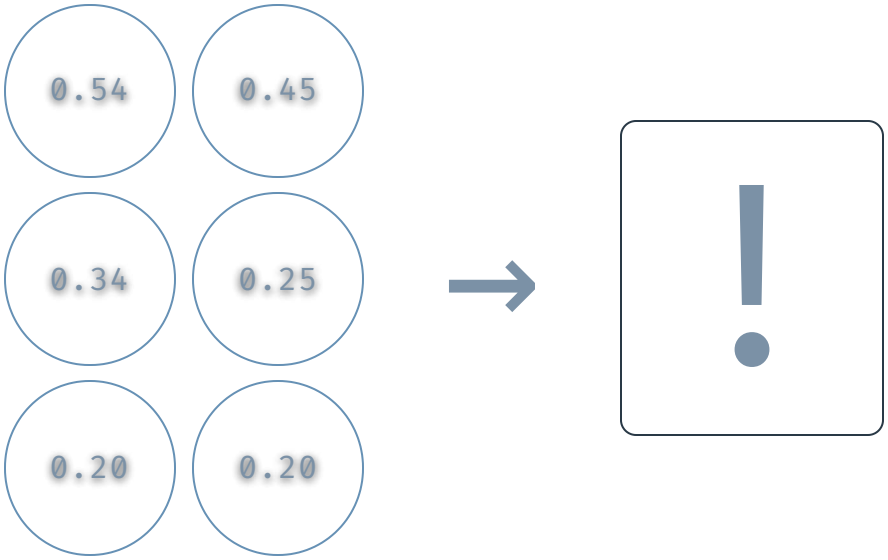
Exponent

So we get:

```
UUUUUUUUYY →
```

As we progress further right, the middle and lower samples get darker, so we get some `f`s:



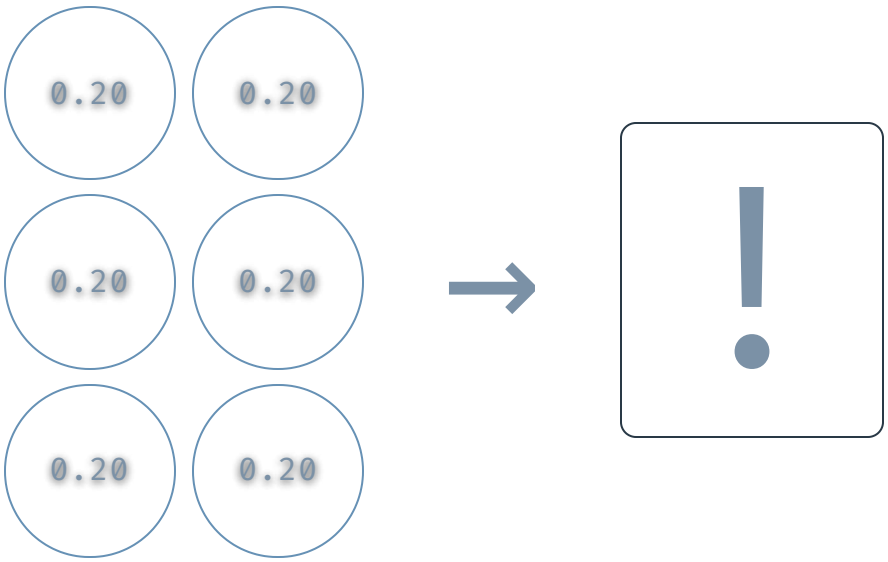


Exponent

Giving us a sequence like so:

```
UUUUUUUUYYf""'''` →
```

That looks good, but at some point we get *no* light samples. Once we get no light samples, our contrast enhancement has no effect because every component is equally light. This causes us to always get `!`s:



Making our sequence look like so:

```
UUUUUUUUYYf""'''`!!!!!!!!!!! →
```

This sudden stop in contrast enhancement having an effect is what causes the staircasing effect:

```
                !!!!!
           !!!!!!!!!!
      !!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!
```

Let's see how we can counteract this staircasing effect with *another* layer of contrast enhancement, this time looking outside of the boundary of each cell.

Directional contrast enhancement

We currently have sampling circles arranged like so:

For each of those sampling circles, we'll specify an "external sampling circle", placed outside of the cell's boundary, like so:

Each of those external sampling circles is "reaching" into the region of a neighboring cell. Together, the samples that are collected by the external sampling circles constitute an "external sampling vector".

Let's simplify the visualization and consider a single example. Imagine that we collected a sampling vector and an external sampling vector that look like so:

The circles colored red are the external sampling vector components. Currently, they have no effect.

The "internal" sampling vector itself is fairly uniform, with values ranging from 0.51 to 0.53. The external vector's values are similar, except in the upper left region where the values are significantly lighter (0.8 and 0.57). This indicates a color boundary above and to the left of the cell.

To enhance this apparent boundary, we'll darken the top-left and middle-left components of the sampling vector. We can do that by applying *component-wise* contrast enhancement using the values from the external vector.

In the previous contrast enhancement, we calculated the maximum component value across the sampling vector and normalized the vector using that value:

```
const maxValue = Math.max( ...samplingVector)

samplingVector = samplingVector.map((value) => {
  value = x / maxValue; // Normalize
  value = Math.pow(x, exponent);
  value = x * maxValue; // Denormalize
  return value;
})
```

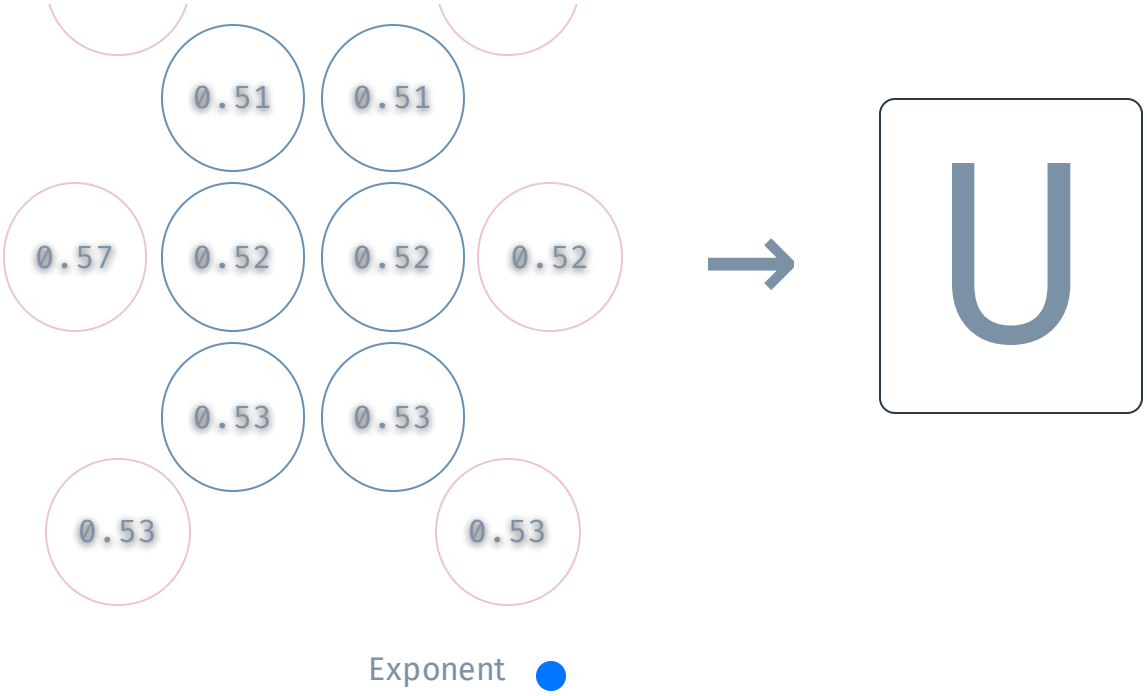
between each component of the sampling vector and the corresponding component in the external sampling vector:

```
samplingVector = samplingVector.map((value, i) => {
  const maxValue = Math.max(value, externalSamplingVector[i])
  // ...
});
```

Aside from that, the contrast enhancement is performed in the same way:

```
samplingVector = samplingVector.map((value, i) => {
  const maxValue = Math.max(value, externalSamplingVector[i]);
  value = value / maxValue;
  value = Math.pow(value, exponent);
  value = value * maxValue;
  return value;
});
```

The example below shows how light values in the external sampling vector push values in the sampling vector down:



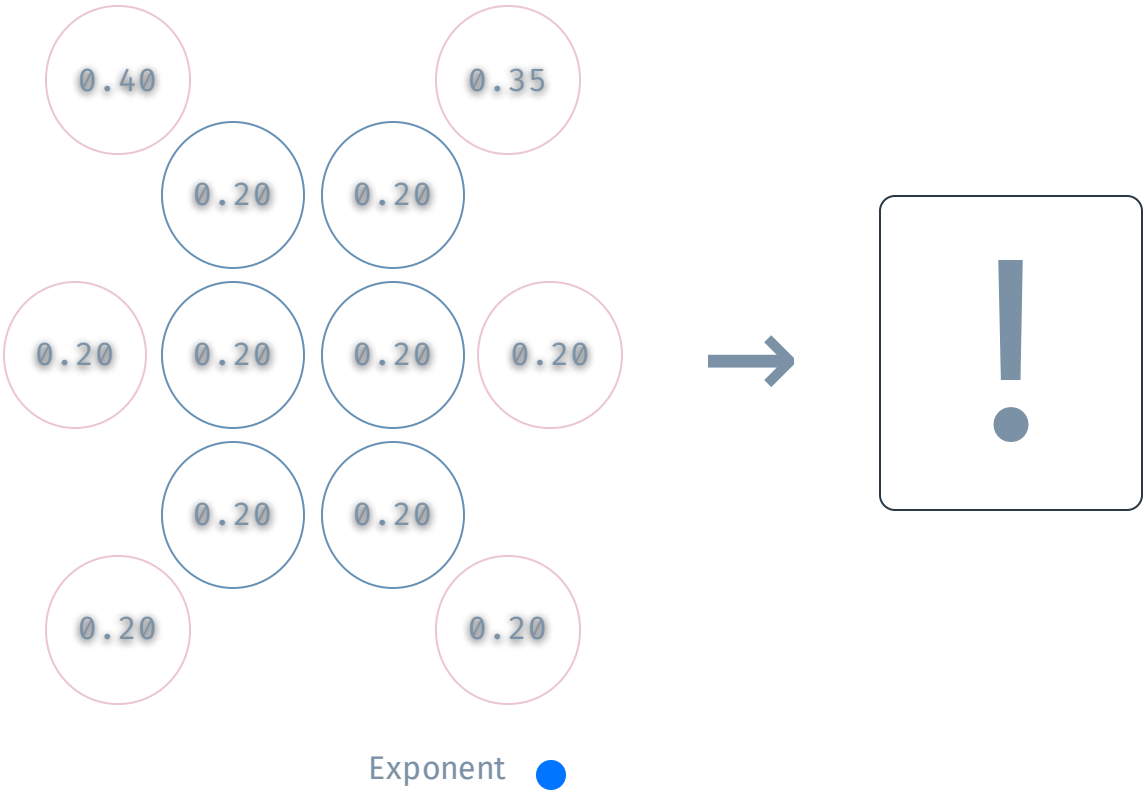
I call this “directional contrast enhancement”, since each of the external sampling circles reaches outside of the cell in the *direction* of the sampling vector component that it is enhancing the contrast of. I describe the other effect as “global contrast enhancement” since it acts on all of the sampling vector’s components together.

Let’s see what this directional contrast enhancement does to get rid of the staircasing effect:

Hmm, that's not doing what I wanted. I wanted to see a sequence like so:

```
          ..::!!
      ..::!!!!!!!!
..::!!!!!!!!!!!!!!!!
```

But we just see `!` changing to `:`



This happens because the directional contrast enhancement doesn't reach far enough into our sampling vector. The light upper values in the external vector *do* push the upper values of the sampling vector down, but because the lightness of the four bottom components is retained, we don't get to `.`, just `:`.

Widening the directional contrast enhancement

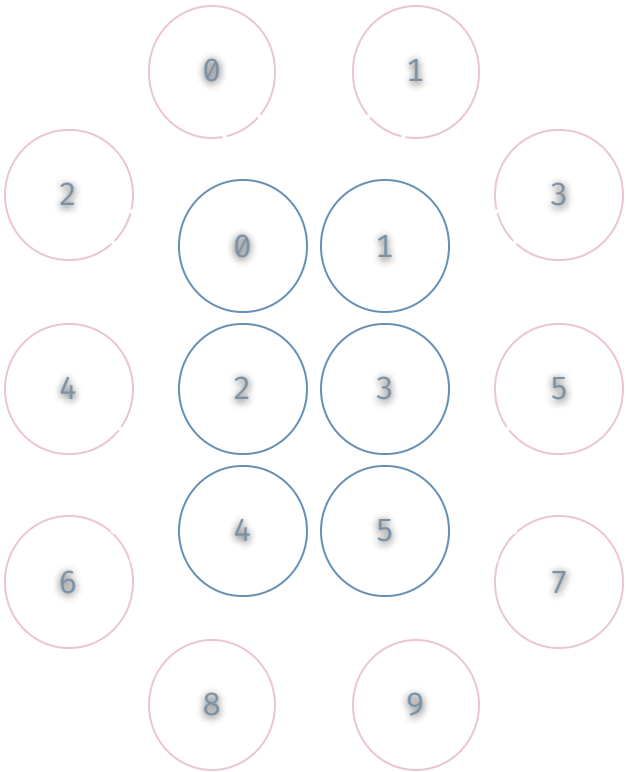
external values at the top spread to the middle components of the sampling vector.

To do that, I'll introduce a few more external sampling circles, arranged like so:

These are a total of 10 external sampling circles. Each of the external sampling circles will affect one or more of the internal sampling circles. Here's an illustration showing which internal circles each external circle affects:

For each component of the internal sampling vector, we'll calculate the maximum value across the external sampling vector components that affect it, and use that maximum to perform the contrast enhancement.

Let's implement that. I'll order the internal and external sampling circles like so:



circles that affect them.

```
const AFFECTING_EXTERNAL_INDICES = [
  [0, 1, 2, 4],
  [0, 1, 3, 5],
  [2, 4, 6],
  [3, 5, 7],
  [4, 6, 8, 9],
  [5, 7, 8, 9],
];
```

With this, we can change the calculation of `maxValue` to take the maximum affecting external value:

```
// Before
const maxValue = Math.max(value, externalSamplingVector[i]);

// After
let maxValue = value;
for (const externalIndex of AFFECTING_EXTERNAL_INDICES[i]) {
  maxValue = Math.max(value, externalSamplingVector[externalIndex]);
}
```

Now look what happens if the top four external sampling circles are light: it causes the contrast enhancement to reach into the middle of the sampling vector, giving us the desired effect:

We now smoothly transition from ! → : → . — beautiful stuff!

Let's see if this change resolves the staircasing effect:

while not being too jagged.

Here's the 3D scene again. The contrast slider now applies both types of contrast enhancement at the same time — try it out:

This really enhances the contrast at boundaries, making the image far more readable!

Together, the 6D shape vector approach and contrast enhancement techniques have given us a really nice final ASCII rendering.

This post was really fun to build and write! I hope you enjoyed reading it.

ASCII rendering is perhaps not the most useful topic to write about, but I think the idea of using a high-dimensional vector to capture shape is interesting and could easily be applied to many other problems. There are parallels to be drawn to word embeddings.

I started writing this ASCII renderer to see if the idea of using a vector to capture the shape of characters would work at all. That approach turned out to work very well, but the initial prototype was terribly slow — I only got single-digit FPS on my iPhone. To get the ASCII renderer running at a smooth 60 FPS on mobile required a lot of optimization work. I describe some of that optimization work in the appendices on character lookup performance and GPU acceleration below.

My colleagues, after reading a draft of this post, suggested *many* alternatives to the approaches I described in this post. For example, why not make the sampling vector 3×3? That would capture the shape of `T` far better — just look how `T`'s stem falls between the two sampling circles in each row:

And yeah, he's right! A 3×3 layout would certainly capture it better. They also suggested many alternative approaches to the contrast enhancement methods I described, but I won't explore those in this post.

There are so, so many approaches and trade-offs to explore. I imagine you probably thought of a few yourself while reading this post!

One dimension I intentionally did not explore was using different colors or lightnesses for the ASCII characters themselves. This is for many reasons, but the two primary ones are that 1) it would have expanded the scope of this post too much, and 2) it's just a different effect, and I personally don't like the look.

At the time of writing these final words, around 6 months have elapsed since I started working on this post. This has been my longest writing process to date. Much of that can be explained by the birth of my now 4-month-old daughter. I've needed to be a lot more intentional about finding time to write — and disciplined when spending it. I intend to write some smaller posts next. Let's see if I manage to stick to that promise.

Thanks for reading! And huge thanks to Gunnlaugur Þór Briem and Eiríkur Fannar Torfason for reading and providing feedback on a draft of this post.

— Alex Harri

Appendix I: Character lookup performance

Earlier in this post, I showed how can find the best character by finding the character with the shortest Euclidean distance to our sampling vector.

```
  let bestCharacter = "";
  let bestDistance = Infinity;

  for (const { character, shapeVector } of CHARACTERS) {
    const dist = getDistance(shapeVector, inputVector);
    if (dist < bestDistance) {
      bestDistance = dist;
      bestCharacter = character;
    }
  }

  return bestCharacter;
}
```

I tried benchmarking this for $100{,}000$ input sampling vectors on my MacBook — 100 K invocations of this function consistently take about 190ms. If we want to be able to use this for an animated canvas at 60 FPS, we only have 16.66ms to render each frame. We can use this to get a rough budget for how many lookups we can perform each frame:

$$100{,}000 \times \frac{16.66\ldots}{190} \approx 8{,}772$$

If we allow ourselves 50% of the performance budget for just lookups, this gives us a budget of about 4K characters. Not terrible, but far from great, especially considering that we're using numbers from a powerful laptop. A mobile device might have a 10 times lower budget. Let's see how we can improve this.

k-d trees

k-d trees are a data structure that enables nearest-neighbor lookups in multi-dimensional (k-dimensional) space. Their performance degrades in higher dimensions (e.g. > 20), but they perform well in 6 dimensions — perfect for our purpose.

Each node can be thought to split the k dimensional space in half with a hyperplane, with the left subtree on one side of the hyperplane and the right subtree on the other.

I won't go into much detail on k-d trees here. You'll have to look at other resources if you're interested in learning more.

One could also look at the hierarchical navigable small worlds (HNSW) algorithm, which Eiríkur pointed me to. It is used for approximate nearest neighbor lookups in vector databases, so definitely relevant.

Let's see how it performs! We'll construct a k-d tree with our characters and their associated vectors:

```
const kdTree = new KdTree(
  CHARACTERS.map(({ character, shapeVector }) => ({
    point: shapeVector,
    data: character,
  }))
);
```

We can now perform nearest-neighbor lookups on the k-d tree:

```
const result = kdTree.findNearest(samplingVector);
```

Running 100K such lookups takes about 66ms on my MacBook. That's about 3x faster than the brute-force approach. We can use this to calculate, roughly, the number of lookups we can perform per frame:

$$100{,}000 \times \frac{16.66\ldots}{66} \approx 25{,}253$$

machine. This is still not good enough.

Let's see how we can eke out even more performance.

Caching

An obvious avenue for speeding up lookups is to cache the result:

```
function searchCached(samplingVector: number[]) {
  const key = generateCacheKey(samplingVector)

  if (cache.has(key)) {
    return cache.get(key)!;
  }

  const result = search(samplingVector);
  cache.set(key, result);
  return result;
}
```

But how does one generate a cache key for a 6-dimensional vector?

Well, one way is to quantize each vector component so that it fits into a set number of bits and packing those bits into a single number. JavaScript numbers give us 32 bits to work with, so each vector component gets 5 bits.

We can quantize a numeric value between 0 and 1 to the range 0 to 31 (the most that 5 bits can store) like so:

```
const BITS = 5;
const RANGE = 2 ** BITS;

function quantizeTo5Bits(value: number) {
  return Math.min(RANGE - 1, Math.floor(value * RANGE));
```

Applying a max of `RANGE - 1` is done so that a `value` of exactly 1 is mapped to 31 instead of 32.

We can quantize each of the sampling vector components in this manner and use bit shifting to pack all of the quantized values into a single number like so:

```
const BITS = 5;
const RANGE = 2 ** BITS;

function generateCacheKey(vector: number[]): number {
  let key = 0;
  for (let i = 0; i < vector.length; i++) {
    const quantized = Math.min(RANGE - 1, Math.floor(vector[i] * RANGE));
    key = (key << BITS) | quantized;
  }
  return key;
}
```

The `RANGE` is current set to `2 ** 5`, but consider how large that makes our key space. Each vector component is one of 32 possible values. With 6 vector components, that makes the total number of possible keys 32^6, which equals $1{,}073{,}741{,}824$. If the cache were to be fully saturated, just storing the keys would take 8GB of memory! I'd also expect the cache hit rate to be incredibly low if we were to lazily fill the cache.

Alright, 32 is too high, but what value should we pick? We can pick any number under 32 for our range. To help, here's a table showing the number of possible keys (and the memory needed to store them) for range values between 6 and 12:

Range	Number of keys	Memory needed to store keys
6	46,656	364 KB

8	262,144	2.00 MB
9	531,441	4.05 MB
10	1,000,000	7.63 MB
11	1,771,561	13.52 MB
12	2,985,984	22.78 MB

There are trade-offs to consider here. As the range gets smaller, the quality of the results drops. If we pick a range of 6, for example, the only possible lightness values are 0, 0.2, 0.4, 0.6, 0.8 and 1. That noticeably affects the quality of character picks.

At the same time, if we increase the possible number of keys, we need more memory to store them. Additionally, the cache hit rate might be very low, especially when the cache is relatively empty.

I ended up picking a range of 8. It's a large enough range that quality doesn't suffer too much while keeping the cache size reasonably low.

Cached lookups are incredibly fast — fast enough that lookup performance just isn't a concern anymore (100K lookups take a few ms on my MacBook). And if we prepopulate the cache, we can expect consistently fast performance, though I encountered no problems just lazily populating the cache.

Appendix II: GPU acceleration

Lookups were not the only performance concern. Just collecting the sampling vectors (internal and external) turned out to be terribly expensive.

Just consider the sheer amount of samples that need to be collected. The 3D scene I've been using as an example uses a 41×107 grid, which equals $4{,}387$ cells. For each of those cells, we compute a 6-dimensional sampling vector and a 10-

compute on every frame.

$$4{,}387 \times (6 + 10) = 70{,}192$$

And that's if we use a sampling quality of 1. If we increase the sampling quality, this number just gets bigger.

Collecting these samples absolutely *crushed* performance on my iPhone, so I needed to either collect fewer samples or speed up the collection of samples. Collecting fewer samples would have meant rendering fewer ASCII characters or removing the directional contrast enhancement, neither of which was an appealing solution.

My initial implementation ran on the CPU, which could only collect one sample at a time. To speed this up, I moved the work of sampling collection and applying the contrast enhancement to the GPU. The pipeline for that looks like so (each of the steps listed is a single shader pass):

1. Collect the raw internal sampling vectors into a $\text{cols} \times \text{rows} \times \text{num circles}$ texture, using the canvas (image) as the input texture.
2. Do the same for the external sampling vectors.
3. Calculate the maximum external value affecting each internal vector component into a $\text{cols} \times \text{rows} \times \text{num circles}$ texture.
4. Apply directional contrast enhancement to each sampling vector component, using the maximum external values texture.
5. Calculate the maximum value for each internal sampling vector into a $\text{cols} \times \text{rows}$ texture.
6. Apply global contrast enhancement to each sampling vector component, using the maximum internal values texture.

I'm glossing over the details because I could spend a whole other post covering them, but moving work to the GPU made the renderer many times more performant than it was when everything ran on the CPU.

To be notified of new posts, subscribe to my mailing list.

Email address Subscribe