CLS

Gia Bamrud

Jun 04, 2023

TUTORIALS

Card Layout Script (CLS) is a language for designing cards. Cards are described as layouts that are rendered by the CLR Renderer.

Use CLS to make cards and things for print-n-play games, prototyping board games, video games, or just to jazz up your favorite TTRPG.

CLS was made because the two main card programs I used both felt lacking. The venerable nanDECK can do most anything, but the opaque syntax and the clinical documentation make it a hard to learn. The WYSIWYG CardMaker is easy to use, but it's limitations felt stifling at times. So I made a language and renderer that did what I wanted it to, and added some other handy features along the way.

Some features of note:

- Powerful placement system allowing you to locate an item from any edge, easily center things, and position things relative to another, all of which is rotation aware
- Units for numbers are built into the syntax, as well as fractions, allowing you to use things like `1/3in` and the renderer knows what you mean
- Rich text in the form of HTML, so you can **bold**, *italicize*, and even include images
- Image and SVG support, the latter allowing you to specify specific items instead of the whole document
- A system of macros that allow you to change any property on any part of a card on a per card basis, with functions for math, string manipulation, and conditions among others
- Cards can be exported to PDF for home printing, textures for Table Top Simulator, or individual images
- Everything is thoroughly documented, and if something isn't clear enough just ask and I'll gladly clarify

CHAPTER

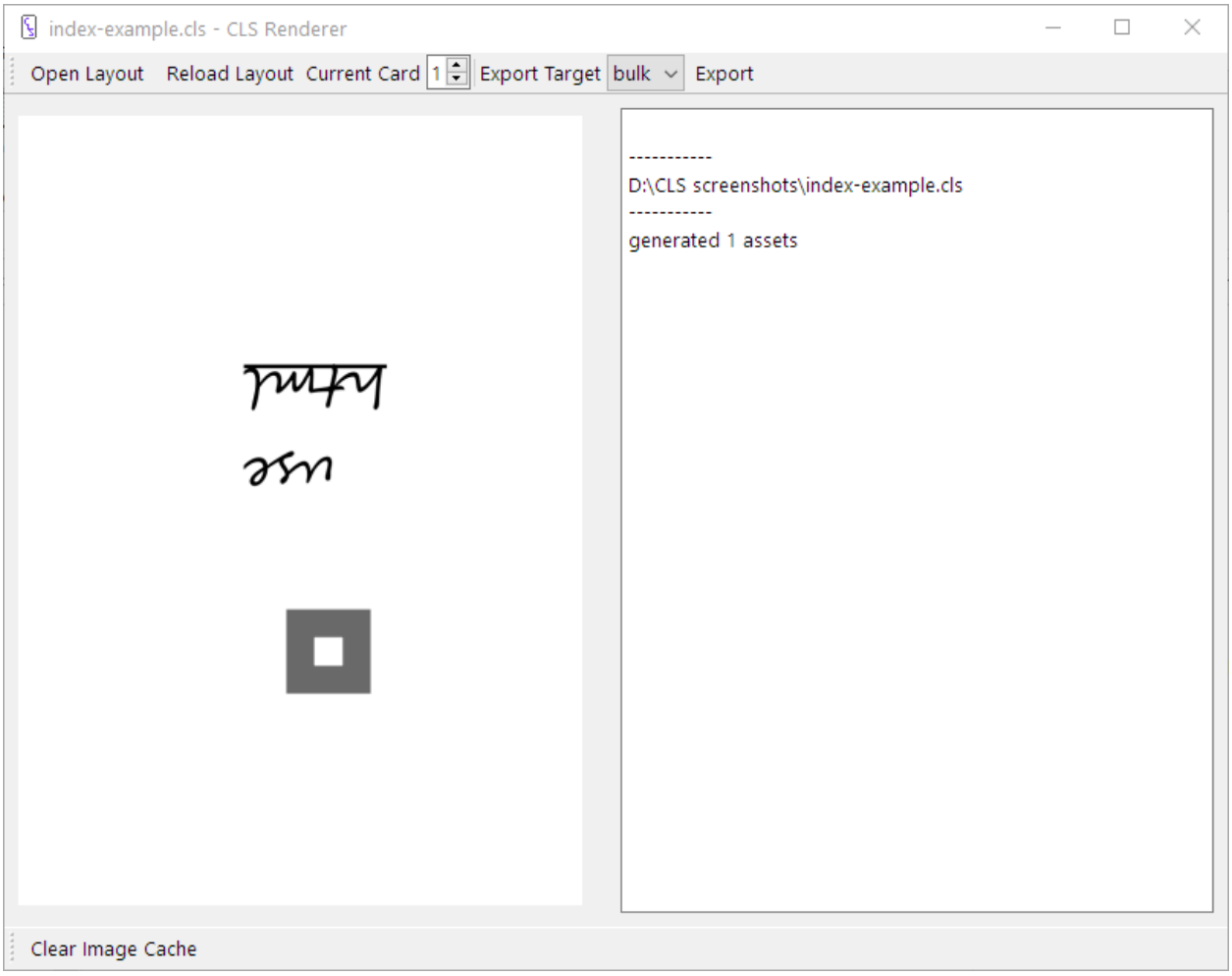
ONE

HOW IT WORKS

Write a layout file like this one

```
layout {
    size: 2.5in, 3.5in
}
label {
    type: text
    position: 1in, 1in
    size: 1in, 1in
    angle: 180
    font: 55pt, Segoe Script
    align: right, bottom
    text: use <u>html</u>
}
box {
    type: rect
    position: ^1in, ^1in
    line: 1/8in, #696969
    fill-color: transparent
}
```

in your favorite editor (there's a plugin for VS Code) and give it to the CLS Renderer to see how it looks.



From there you can export your cards to individual images, a texture image for Tabletop Simulator, or a PDF for print and play.

Download the CLR Renderer from the home page at codlark.itch.io/CLS then take a look at the *tutorial*.

1.1 Getting Started

This tutorial will explain the basics of CLS by working through a simple set of role cards for the game Werewolf.

To get started, download and install the CLS Renderer, and open up your favorite code editor. If you don't already have a favorite code editor, I recommend VS Code, by Microsoft. There's an official language plugin for VS Code that'll make it easier to write layouts, just look for it in the extension marketplace.

Tip: If you've never programmed before I recommend typing out each example as we find it. Programming makes use of lots of special characters and it can take time to build the habits for typing them.

1.1.1 Layout Files

CLS is used to make layout files with the extension ".cls" to describe a generalized card (the layout), which gets turned into a set of actual cards. Layouts are made up of sections, which are split into special sections and elements. Elements are the individual elements of the card, like a text box or an image. Special sections do all sorts of things, like setting the size of the card, changing the name of exported pdf files, and specifying data used to render cards.

Let's start with the `layout` special section.

1.1.2 The Layout Section

The basic structure of a section is: a name, a pair of curly braces, and in those braces, content. Most commonly that content is in the from of properties which are: name, a colon, and one or more values separated by commas.

```
layout {
    size: 2.5in, 3.5in
}
```

The `layout` section sets the size and resolution of the card as well as how to process the data. The size of this card is 2.5 inches wide and 3.5 inches tall. This is the standard US Poker card size. We're using the default resolution of 300 dpi (dots per inch) which is a commonly used resolution for printing. As for the data…

1.1.3 The Data Section

When the CLS Renderer process a layout it combines the layout with a data table one row at a time, and each row creates a card. We could use an external editor like Excel to create this data, but for small sets it's easier to embed it right in the layout file with the `data` section.

```
data {
repeat, role
2, werewolf
1, seer
4, villager
}
```

Data is represented as comma separated values, CSV for short.

The first row acts as the names of macros used to reach the data. The column `repeat` is a special column that causes that row to generate that many cards, so there'll be 2 werewolf cards, 1 seer card, and 4 villager cards.

Data can be represented two ways, the first as the `data` section shown above or as a property of the `layout` section as described *here*. Unlike the other sections, data is not indented.

1.1.4 The Macros Section

Macros are how values are pulled from the data, these are called variables. They can also modify data and select values conditionally, these are called functions. The `macros` special section lets us make our own macros, like this color variable.

```
macros {
    dark-red = #a32b1d
}
```

Notice that macros are defined with an equal sign. You can also make your own functions, which is described in more detail *here*.

Colors in CLS use the standard hex color format, either #RRGGBB or #AARRGGBB if you want transparency. There a hand full of common color names available, such as `black`, `white`, and `transparent`; but outside of these three making your own colors will generally look better.

Now onto putting things on the cards.

1.1.5 Element Sections

Elements are parts of a layout, these can be text, images, or simple shapes. Each element is it's own section, and as we'll see in the next chapter, elements can contain other elements.

1.1.6 The Text Element

The first thing we should put on a card is the role, so let's look at a text element.

```
role {
    type: text
    position: center, .5in
    size: 1.5in, .25in
    font: 12pt, Palatino Linotype
    font-color: [if| [eq| [role], werewolf], [dark-red], black]
    align: center, middle
    text: [capitalize| [role]]
}
```

That looks like a lot, but let's break it down so we can see what it's all doing.

```
role {
    type: text
```

This creates the element, which will be named "role", and be of type `text`, which as you may guess is an element type that draws text.

```
    position: center, .5in
    size: 1.5in, .25in
```

These lines position and size the element. We want it centered horizontally and that's exactly what the `center` keyword does, then a half inch away from the top of the page. The size is one and a half inches wide and a quarter inch tall. We could also use fractions for the same effect, as in

```
    position: center, 1/2in
    size: 1 1/2in, 1/4in
```

Fractions and decimals are equivalent, some numbers are easier in one or the other, like `1/8` or `0.3`. For this tutorial we'll be using decimals everywhere.

```
    font: 12pt, Palatino Linotype
    font-color: [if| [eq| [role], werewolf], [dark-red], black]
    align: center, middle
```

The `font` property sets the size and name of the font, and optionally the color of the font. For readability I've split the color out. The `align` property positions the text within its defined size. As for that `font-color`, it's our first look at macros and we're gonna take it inside out.

```
[role]
```

This is easy, the `[role]` variable pulls the value of the `role` column for this card/row.

```
[eq| [role], werewolf ]
```

Functions use vertical bars to separate the name of the function from it's arguments, and commas to separate arguments from each other. Whenever you see a vertical bar in a macro it means it's performing extra computation on it's value. In the case of the `[eq| ]` function here, if its arguments are the same, it returns (turns into) `true`, or otherwise returns `false`.

```
font-color: [if| [eq| [role], werewolf ], [dark-red], black]
```

This is the whole value. The `[if| ]` functions takes as it's first value a true or false value, called a toggle in CLS, and if it's true it returns the second argument, or the third argument it's false. So this whole thing checks if the role of the current card is `werewolf`, and if it is the font color is made `[dark-red]`, or if it's some other role the font color is made `black`. There are a fair number of macros which are listed *here*.

```
    text: [capitalize| [role]]
```

Lastly, the text that will actually be drawn on the card. Compared to our last macro, this is pretty easy; it just capitalizes the first letter of `[role]`.

In your own layouts you'll see macro sequences like this just as often as the value for `font-color`. Macros are a powerful programming tool, but never feel like you have to find the most elaborate macro sequence to do the most with the fewest columns. I only use a complex macro sequence to show you what's possible with macros. We could just as easily have data that has a color column like

```
data {
repeat, role, color
2, werewolf, [dark-red]
1, seer, black
4, villager, black
}
```

You can see the color macro we defined earlier right in the data. macroWork can tell when macros return other macros and will process them until no macros are left. Then for `font-color` all we need is

```
font-color: [color]
```

This would work just as well.

1.1.7 The Image Section

Theoretically, we could stop here, all we need to do is tell players what role they have. But let's give them an icon.

```
icon {
    type: image
    position: center, 1in
    source: images/[role].png
}
```

This should all make sense by now. The only new thing is the `source` property that tells us where the image is located. We don't need to specify a size because CLS will use the image at full size by default.

These images come with the CLS Renderer in the examples folder.

1.1.8 The Export Section

The last step before we can make any cards is the `export` section.

```
export {
    destination: cards
    bulk {
        name: [role][repeat-index].png
    }
}
```

The `export` section operates on export targets, each of which specifies settings for one method of exporting cards. There are currently three export targets in CLS:

- `bulk` exports cards to their own images
- `pdf` exports the cards to a pdf
- `tts` exports the cards to images suitable for Tabletop Simulator

Properties within the `export` section directly change the specified setting for all export targets, so here everything will be exported to a folder named "cards" that lives in the same folder as the layout file. If this folder does not exist the CLS Renderer will make it.

The `export` section can also have a subsection named for each export target, and the properties in those subsections control that export target only. So in this case the `bulk` export target is set to name each card according to it's role value and `[repeat-index]`, which is the number of times this row has been used to make a card.

A full exploration of the `export` section can be found *here*

1.1.9 First Look at the Cards

Let's take a look at the cards, but first a look at the layout file all together

```
layout {
    size: 2.5in, 3.5in
}

export {
    output: cards
```

(continues on next page)

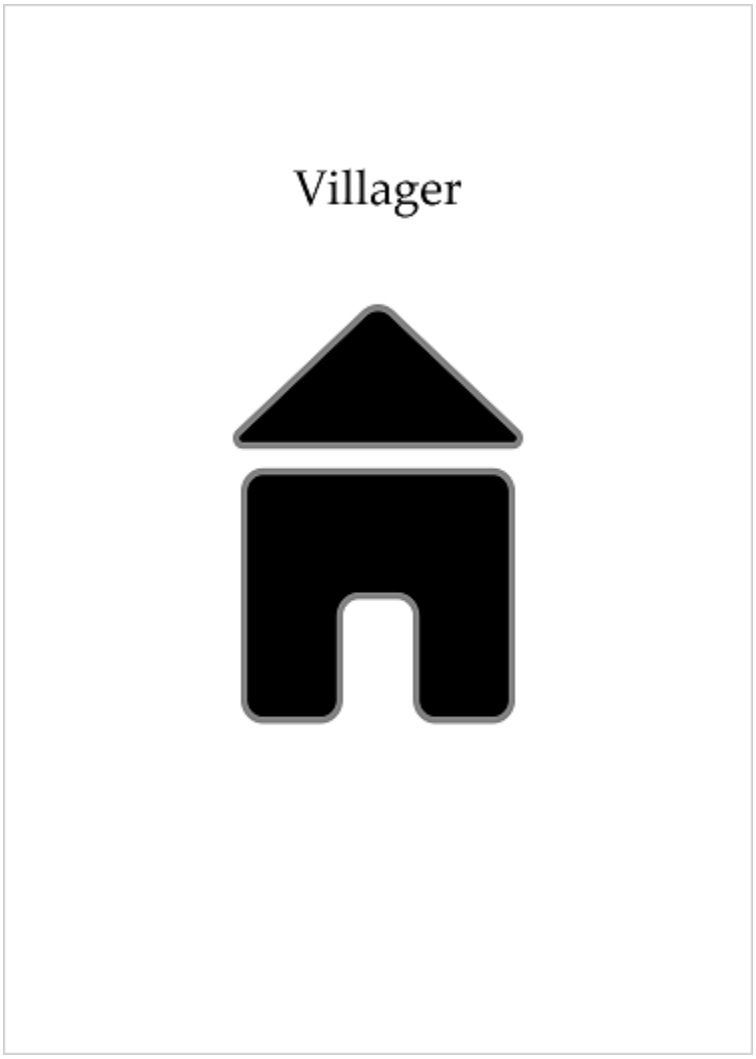
(continued from previous page)

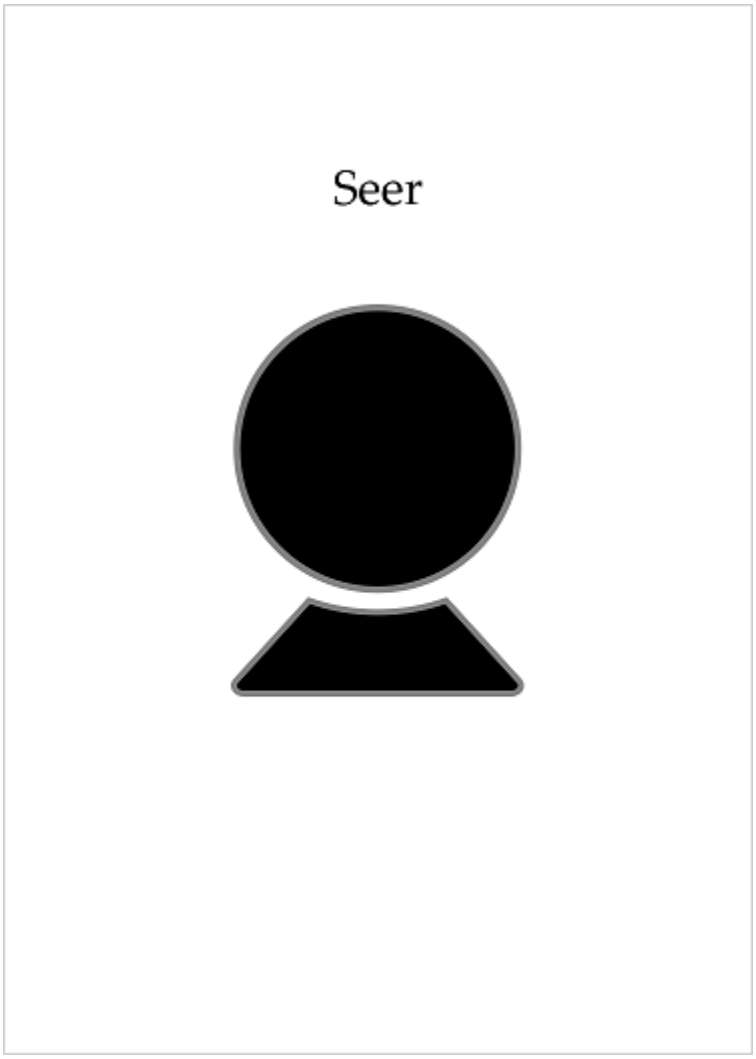
```
    bulk {
        name: [role][repeat-index].png
    }
}

macros {
    dark-red = #a32b1d
}

role-border {
    type: rect
    position: center, .5in
    size: 1.5in, .25in
    corner-radius: .1in
}

role {
    type: text
    position: center, .5in
    size: 1.5in, .25in
    font: 12pt, Palatino Linotype
    font-color: [if| [eq| [role] | werewolf ] | [dark-red] | black ]
    align: center, middle
    text: [capitalize| [role] ]
}

icon {
    type: image
    position: center, 1in
    source: images/[role].png
}

data {
repeat, role
2, werewolf
1, seer
4, villager
}
```

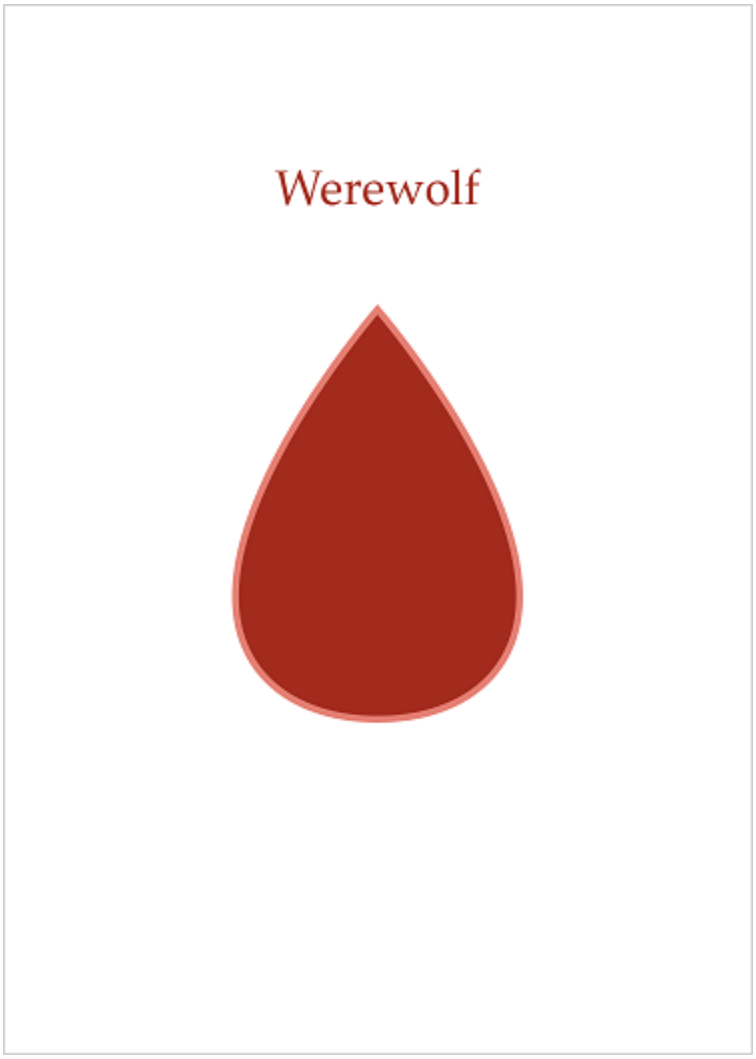
You'll notice that the sections aren't in the order that we first saw them in! Sections follow a canonical order to make reading layout files easier:

- `layout`
- `export` the subsections can be in any order
- `macros`
- `defaults` we'll see this section on the next page!
- element sections in reading order; left to right, top to bottom
- `data`

So if we open this in the CLS Renderer and click the Export button we'll get a folder named cards with seven cards in it that look like these.


Villager


Seer



Note: These cards were given a thin border to make it easier to see their size, your cards will not have this border.

Looks pretty good but we make them better. In the next section of this tutorial we'll look at some more elements types and advanced features.

1.2 Werewolf Part 2

In chapter 1 we looked at a simple layout for a set of cards for Werewolf. This chapter will continue working on those cards.

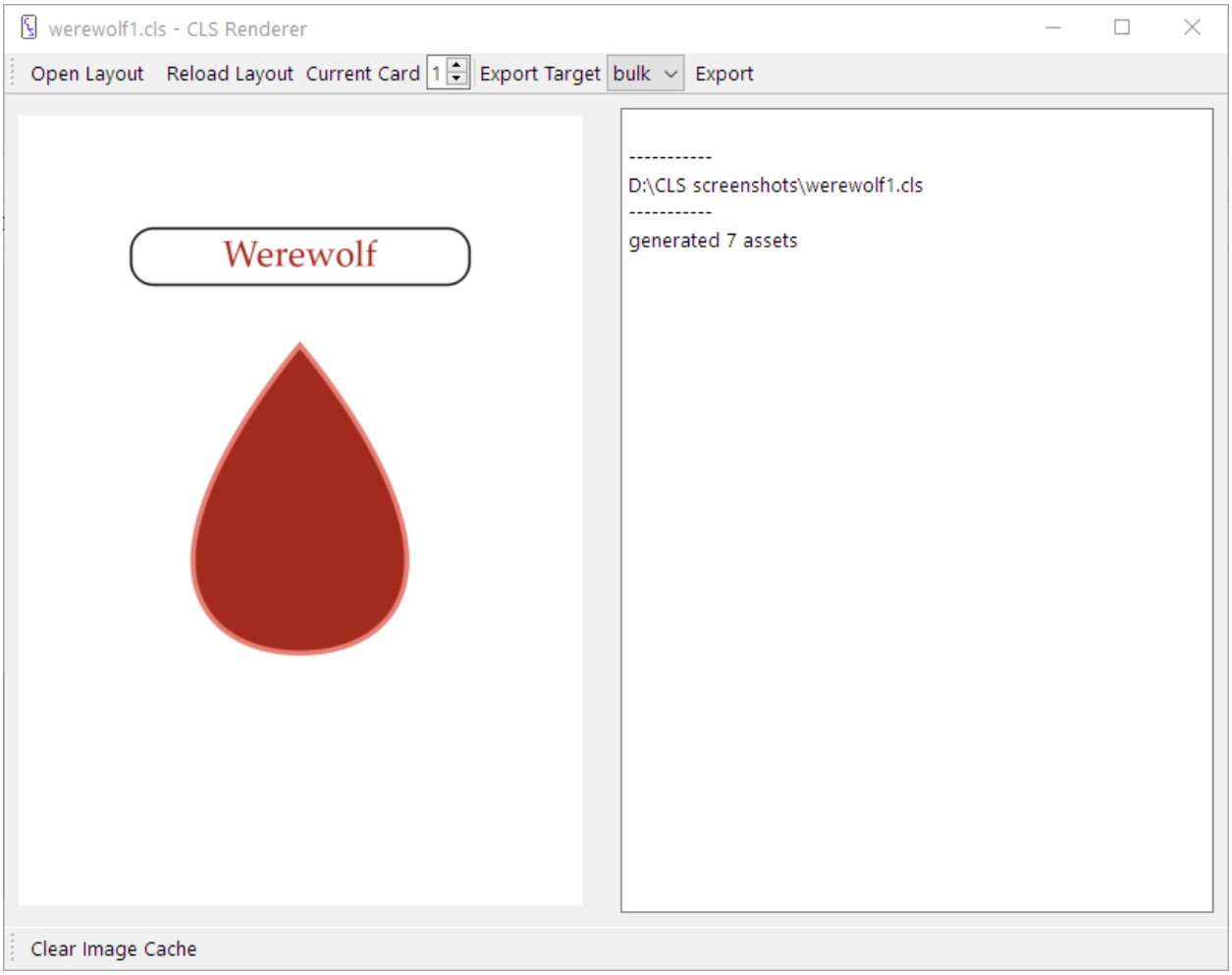
1.2.1 The Rect Element

CLS offers some limited elements for drawing shapes. We'll be looking at `rect` but there's also `circle` and `line`. Shapes are great for placeholders until you get final graphics, simple layouts, or colorizing transparent images. The color of the shape and the line used to draw the shapes are also fairly flexible, check out the section on *Element Shapes* for more.

So, let's look at a simple border.

```
role-border {
    type: rect
    position: center, .5in
    size: 1.5in, .25in
    corner-radius: .1in
}
```

Pretty easy! We give it the same position and size as the role element so it fits, and the `corner-radius` property rounds the corners slightly. Just put this **before** the role section and let's take a look



Looks good, but there's a problem. Because we put the same size and position in two places, if we want to move the role we have to change things in two places. We can fix that by putting the elements in a container.

```
role-container {
    position: center, .5in
    size: 1.5in, .25in

    border {
        type: rect
        size: 100%, 100%
        corner-radius: .1in
    }

    role {
        type: text
        size: 100%, 100%
        font: 12pt, Palatino Linotype
        font-color: [if| [eq| [role], werewolf], [dark-red], black]

        align: center, middle
        text: [capitalize, [role]]
    }
}
```

A container is any element that holds another element, as shown. An element in another element is a subelement. If an element has no other container then the layout can be thought of its container. Any element can be a container, but elements without types are recommended. Containers affect their subelements in a few ways

- The position of a subelement is based on the location and rotation of its container.
- When the size of a subelement is given as a percent its size is proportional to its container
- Subelements are drawn after their container.

This gives us the benefit of separation of concerns, which is programmer talk for "different things in different places, so separate things can do the same thing". You don't have to put everything in a container, but when you have multiple elements of the same size at the same place containers certainly help. Other places you might want to use a container.

- position one element relative to another.
- a set of elements that will always be next to each other, but don't have a final position
- you want to define a size as a percent of another element's size

1.2.2 Inverse Position

While we're on the subject of positioning, let's talk inverse positions. Normally, positions are based on the upper left, where a given element's upper left is relative to its container's upper left. Some times you want to position an element based on its lower or right edge. This is the purpose of inverse positioning. We'll look at this by adding a description of the role to the bottom of the cards.

```
description {
    type: text
    position: center, ^.5in
    size: 2in, .5in
    font: 10pt, Palatino Linotype
    align: center, middle
    text: [description]
```

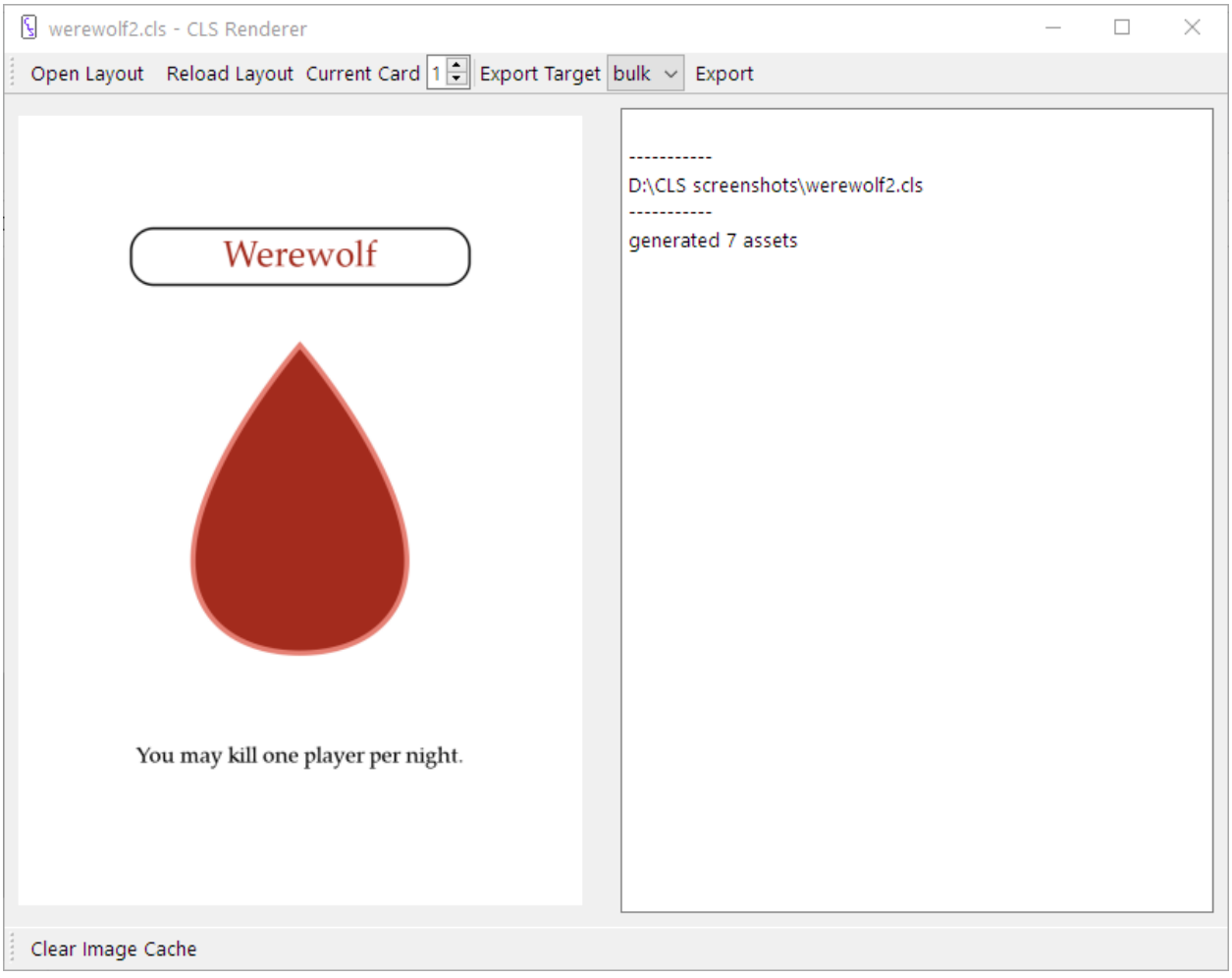
(continues on next page)

(continued from previous page)

```
}

data {
repeat, role, description
2, werewolf, You may kill one player per night.
1, seer, You may find out the role of one player per night.
4, villager, You have no abilities.
}
```

An inverse position is indicated with a caret before the number. In this case the description will have it's bottom a half inch from the bottom of the card. Inverse Positions can be mixed with normal positions to specify any corner, such as `position: ^.25in, 25in` to specify a quarter inch from the upper right corner.

Let's take a look then we'll go over a few final points.



Neat!

1.2.3 The Defaults Section

Because our two text boxes use the same font, we can make use of the `defaults` section.

```
defaults {
    font-family: Palatino Linotype
}
```

The `defaults` section sets default values for every element in the layout. Any property can be defined in the `defaults` section, even size and position. This is handy any time multiple elements use the same value and you only want to define it in one spot.

1.2.4 Image Positioning

Images can be a little weird to position. We have a simple example and our images are about the same size so there isn't much to do. For more complex positioning of images of different sizes, use an `image-box` element. It works like an `image` element, with an added `align` property paired with a defined size to position images within that size. Check it out *here* to familiarize yourself with it, and how the `image` element works.

1.2.5 Last Look at the Cards

Finally, here's everything all together.

```
layout {
    size: 2.5in, 3.5in
}

export {
    destination: cards
    bulk {
        name: [role][repeat-index].png
    }
}

macros {
    dark-red = #a32b1d
}

defaults {
    font-family: Palatino Linotype
}

role-container {
    position: center, .5in
    size: 1.5in, .25in

    border {
        type: rect
        size: 100%, 100%
        corner-radius: .1in
    }
```

(continues on next page)

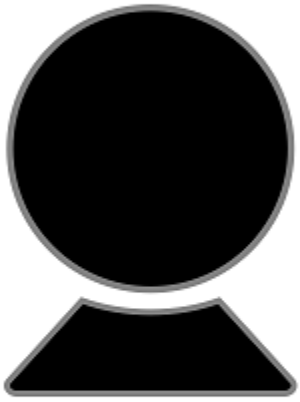
(continued from previous page)

```
    role {
        type: text
        size: 100%, 100%
        font-size: 12pt
        font-color: [if| [eq| [role], werewolf], [dark-red], black]

        align: center, middle
        text: [capitalize, [role]]
    }
}

icon {
    type: image
    position: center, 1in
    source: images/[role].png
}

description {
    type: text
    position: center, ^.5in
    size: 2in, .5in
    font-size: 10pt
    align: center, middle
    text: [description]
}

data {
repeat, role, description
2, werewolf, You may kill one player per night.
1, seer, You may find out the role of one player per night.
4, villager, You have no abilities.
}
```

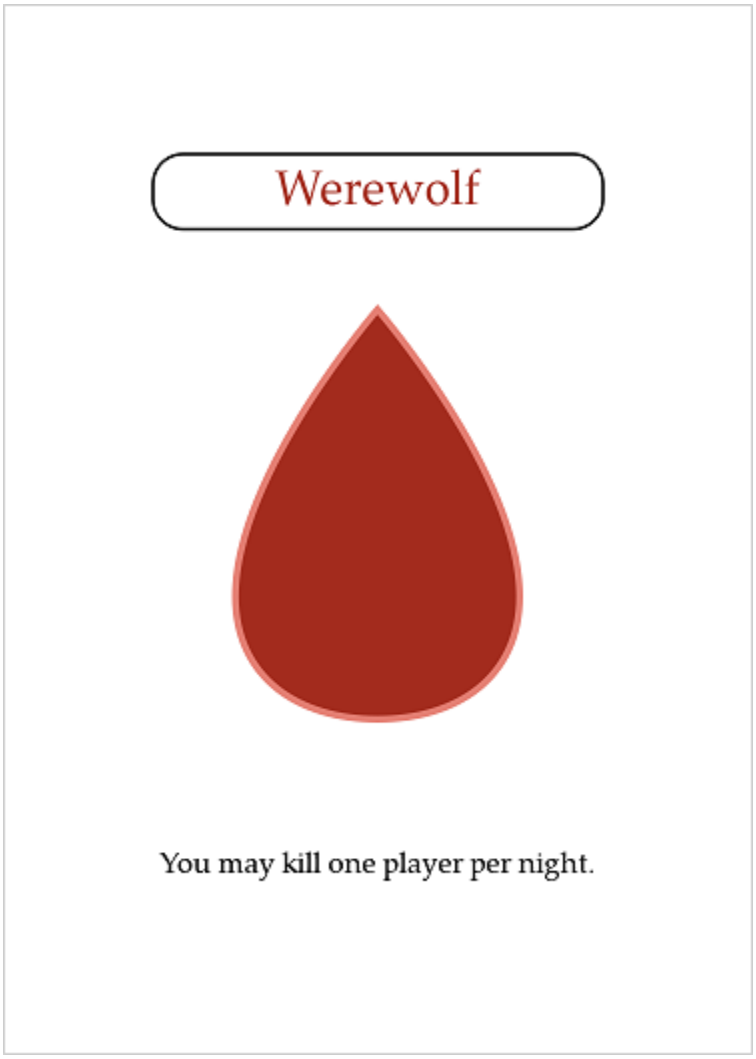
And the cards


Villager
You have no abilities.

Seer



You may find out the role of one player per night.



Neat! This final card is included with the CLS Renderer in the example folder, so take a look at that if you haven't been following along.

Warning: The example included with the renderer includes an issue, specifically the font sizes are too big. Until that's fixed, you should copy and paste the code above.

1.3 Example - Playing Cards

On this page we're going to look at making a deck of playing cards, including jokers. We'll look at when to use templates, and how to use them. This guide is less a tutorial, and more of a walkthrough. Just like with the Werewolf example before, all the images used and layout files are provided with CLS.

1.3.1 Deck Structure

Templates are used whenever multiple card layouts use the same or similar properties and elements. In the case of playing cards this means the indexes and the special sections. Then the aces, ranks, face cards, and jokers each use a separate layout that uses the index layout as a template.

1.3.2 Indexes

Because it's going to be the first thing the CLS Renderer sees, let's look at indexes.cls

```
layout {
    size: 2.5in, 3.5in
}
export {
    destination: cards/
    bulk {
        name: [suit][rank].png
    }
}
macros {
    red = #ff1569
    black = #1a1a5e
    rank = 0
}
upper-left-index {
    position: 1/8in, 1/8in
    size: .3in, .6in

    number {
        type: text
        size: .3in, .3in
        font: 1/4in, Consolas
        font-color: [if| [in| [suit], spade, club], [black], [red]]
        align: center, top
        text: [rank]
    }

    pip {
        type: image
        position: center, .3in
        source: images/[suit]-small.png
    }
}

lower-right-index {
```

(continues on next page)

(continued from previous page)

```
    position: ^1/8in, ^1/8in
    size: .3in, .6in
    angle: 180

    number {
        type: text
        size: .3in, .3in
        font: 1/4in, Consolas
        font-color: [if| [in| [suit], spade, club], [black], [red]]
        align: center, top
        text: [rank]
    }

    pip {
        type: image
        position: center, .3in
        source: images/[suit]-small.png
    }
}

data{
suit
spade
heart
club
diamond
}
```

There are actually quite a few design choices in selecting size and position of the indexes, but beyond that

- we define two variables, `[black]` and `[red]`, set to the colors of the pips, we use these for the font color of the rank
- a third variable, `[rank]` is defined and set to 0. This is more of a placeholder on this layout, and will get filled in by each layout
- the lower right index uses caret positioning and is rotated, but otherwise is the same as the upper left index
- the data section only features the suits, this is because we can to replace it on every

1.3.3 Aces

The aces are pretty simple because they can inherit so much from the index layout, let's see how little aces.cls does

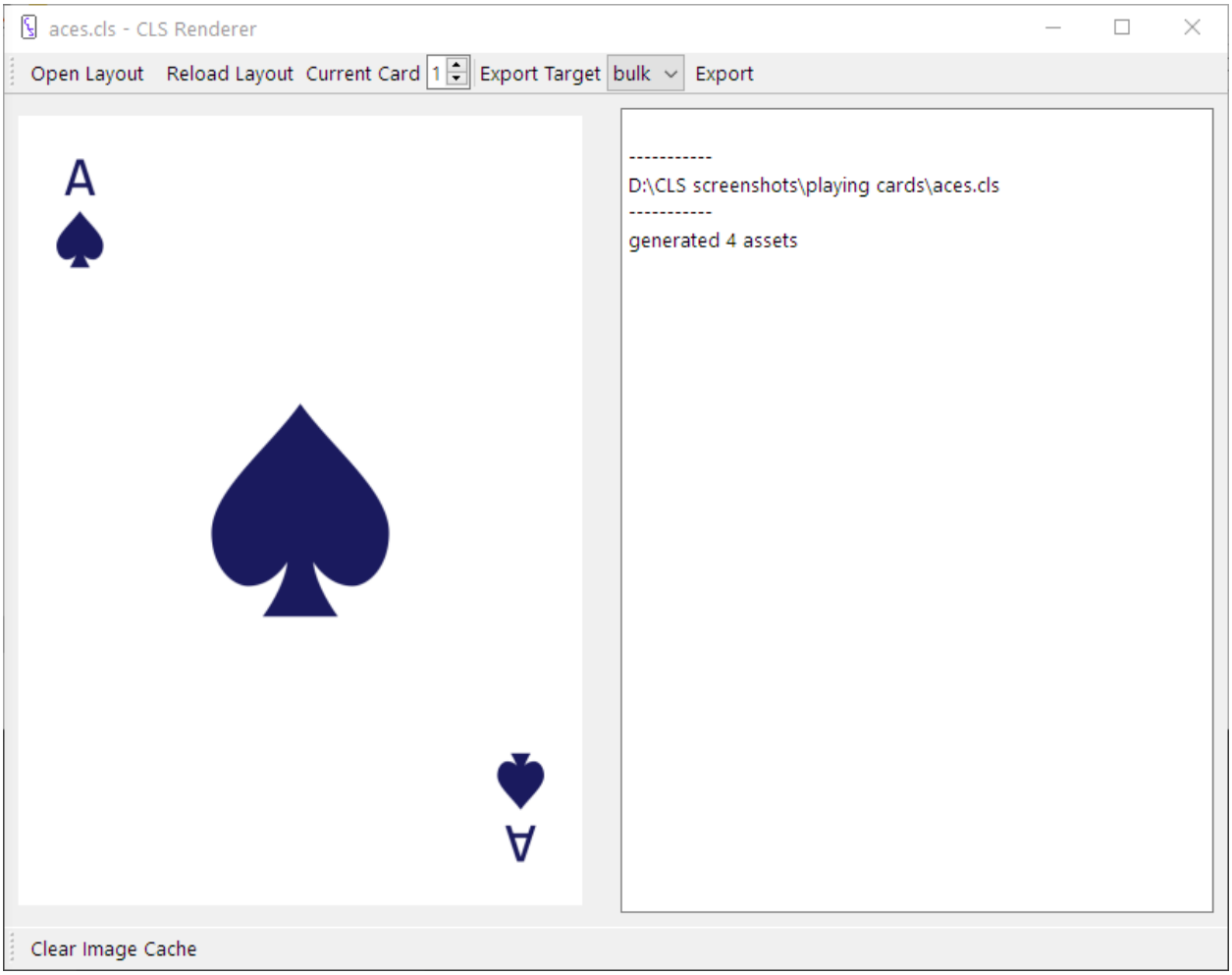
```
layout {
    template: indexes.cls
}

macros {
    rank = A
}

ace {
```

(continues on next page)

(continued from previous page)

```
    type: image
    position: center, center
    source: images/[suit]-big.png
}
```

There really isn't a lot there, is there? We define indexes.cls as our template, set `[rank]` to `A` and place the image. We don't even have to use different data. Let's take a look at this now.



Where the Images Come From

As an aside, the art for these cards is pulled from a couple different projects I've worked on in the past. Also, the different size pips are tweaked to look better at their specific size.

1.3.4 Ranks

The ranks do a lot in their layout, so I'm only going to show parts of it

```
layout {
    template: indexes.cls
}
macros {
    rank = [=| [repeat-index] + 1]
}

defaults {
    source: images/[suit].png
}

row1 {
    position: .5in, .5in
    size: 1.5in, .5in
    draw: [in| [rank], 8, 9, 10]
    col1 {
        type: image
    }
    col3 {
        type: image
        x: ^0
    }
}
...
row4 {
    position: 0.5in, center
    size: 1.5in, .5in
    col1 {
        type: image
        draw: [in| [rank], 6]
    }
    col2 {
        type: image
        x: center
        draw: [in| [rank], 3, 5, 7, 9]
    }
    col3 {
        type: image
        x: ^0
        draw: [in| [rank], 6]
    }
}
...
row7 {
    position: 0.5in, ^.5in
    size: 1.5in, .5in
    angle: 180
    draw: [in| [rank], 8, 9, 10]
    col1 {
```

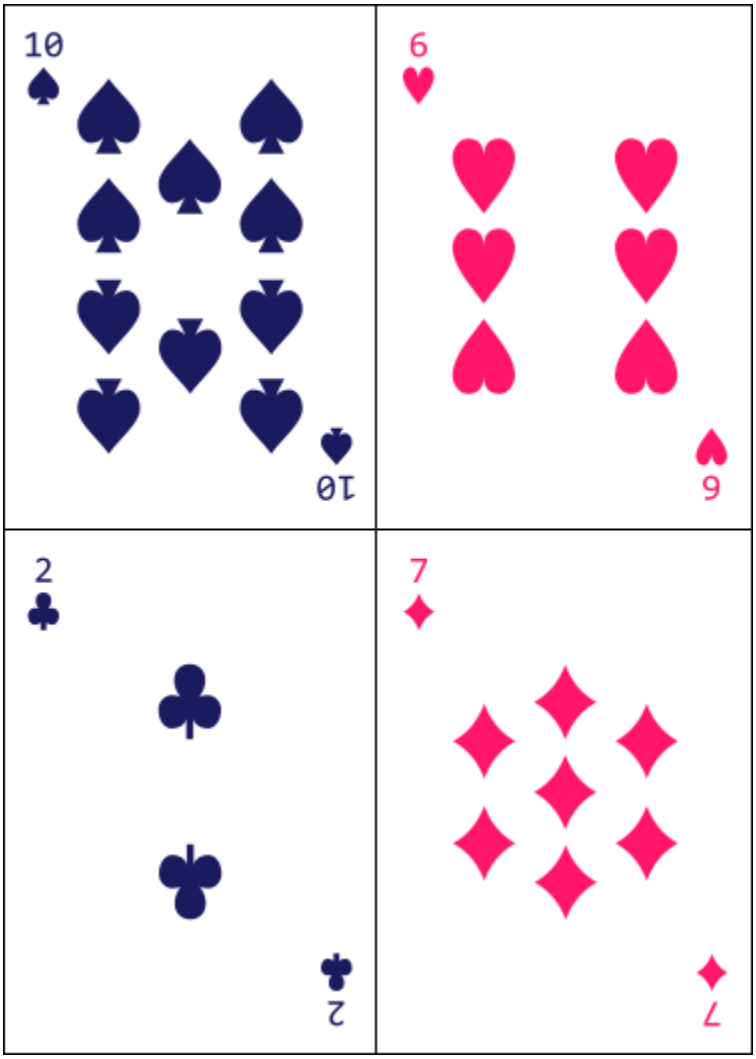
(continues on next page)

(continued from previous page)

```
        type: image
    }
    col3 {
        type: image
        x: ^0
    }
}

data {
repeat, suit
9, spade
9, heart
9, club
9, diamond
}
```

I wasn't kidding when I said this did a lot! We use indexes.cls as a template, but that's about the only thing it does the same as the aces. You'll see `[rank]` is defined as adding one to `[repeat-index]`, that way we can use a repeat column without having to skip 1 or ignore it in the output. In the `defaults` section we define the default image to load as the pip of the current suit so we don't have to put it 10 different places.

The pips are were things get interesting. We define them a row at a time, then in each row we name them by column. Then we draw them based on their rank by checking to see if the pip should be seen for this rank. You can see how caret positioning and `center` make it simple to position everything.



I've shrunk the cards down some, but you can see how the pips get laid out properly.

1.3.5 Face Cards

```
layout {
    template: indexes.cls
}

macros {
    rank = [substr| [royal], 1, 1]
}

portrait {
    type: image
    position: center, center
    source: images/[color][royal].png
}
```

(continues on next page)

(continued from previous page)

```
data {
suit, color, royal
spade, black, Jack
spade, black, Queen
spade, black, King
heart, red, Jack
heart, red, Queen
heart, red, King
club, black, Jack
club, black, Queen
club, black, King
diamond, red, Jack
diamond, red, Queen
diamond, red, King
}
```

Other than `[rank]`, which is set to the first character of `[royal]`, there isn't anything here we haven't seen before. I'm not going to put a picture of any face cards here, as you can probably assume what it looks like.

1.3.6 Jokers

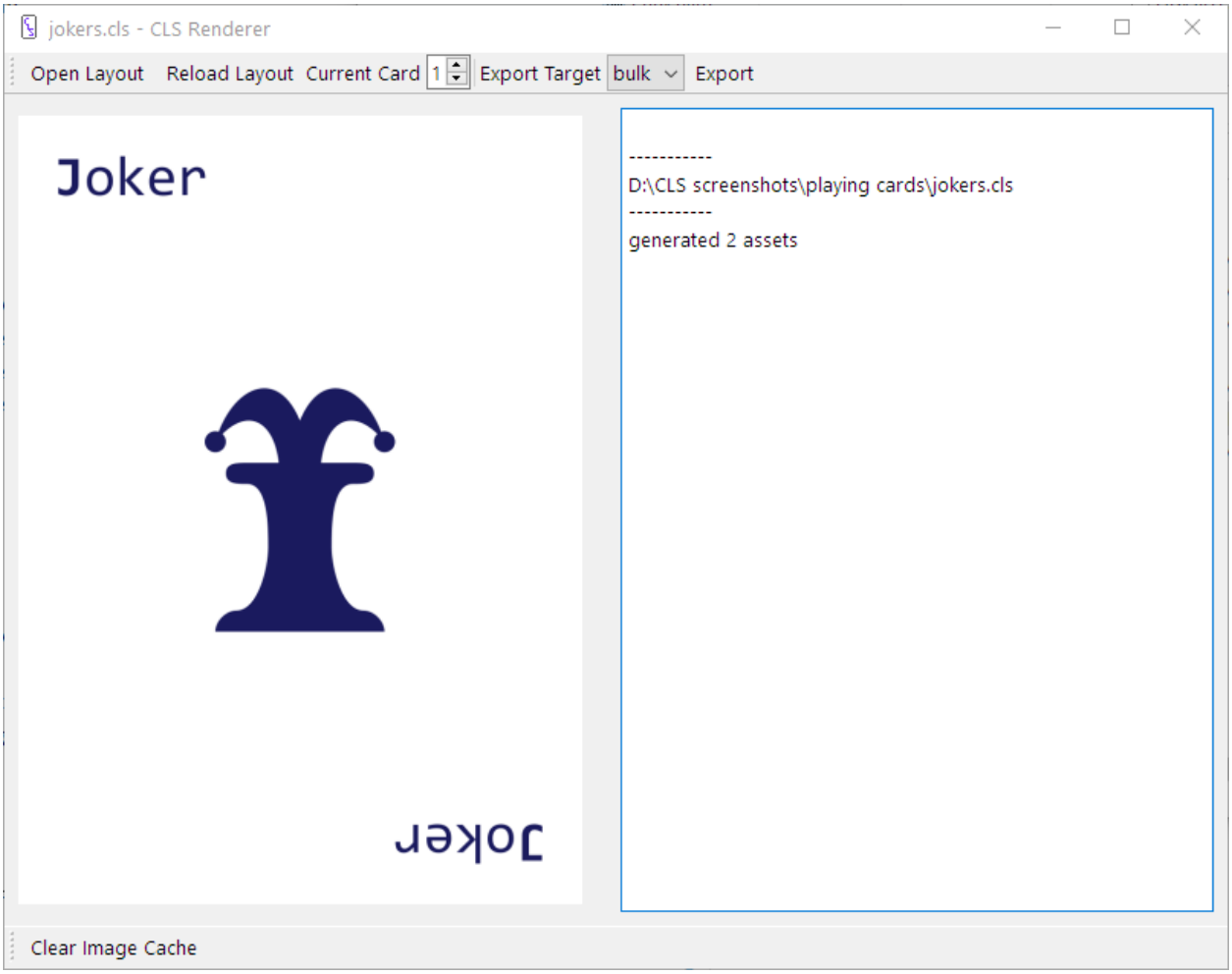
```
layout {
    template: indexes.cls
}

export {
    bulk {
        name: [name].png
    }
}

upper-left-index {
    number {
        size: .75in, .5in
        font-color: [color]
        align: center, top
        text: [b|J]oker
    }
    pip {
        draw: off
    }
}

lower-right-index {
    number {
        size: .75in, .5in
        font-color: [color]
        align: center, top
        text: [b|J]oker
    }
    pip {
```

(continues on next page)

(continued from previous page)

```
        draw: off
    }
}

portrait {
    type: image
    position: center, center
    source: images/[name].png
}

data {
color, name
[black], bigJoker
[red], smallJoker
}
```

The jokers do quite a bit. The big thing is the way they modify the indexes. Because they don't have a true rank and no suit, they don't need the standard index, but the position is still good. So they turn off the pip index and change the number index to show Joker, like so



The way this works is that when you load in a template, the special sections, elements, properties, and data that belong

to the template are all put in memory. When the main layout gets loaded, that layout's elements, properties, etc. get put in the exact same place, overwriting anything that's there, and just as importantly, leaving anything that the layout doesn't change untouched. So when jokers.cls changes the `text` property without touching `position`, the `position` set in indexes.cls is kept.

And with that we have a deck of 54 cards and hopefully a better understanding of how templates work.

1.4 Example - Making a Card Game

There are several card makers that specialize in Magic the Gathering cards so there's little need to show you how to make MtG cards with CLS. On the other hand, MtG cards feature many things that people put in their own card games, like complex costs, inline images, and flavor text. So let's take a look at "Alchemy the Collecting", a game that totally isn't Magic the Gathering.

These cards were thrown together for this example with stock are for the main image.

1.4.1 Data

Let's look at the data, which will help the rest of the layout make sense.

```
data {
title, frame, art, cost, type, effect, flavor
Stone Wall, earth, stone-wall.jpg, (any, earth), Continuous, Pay [text-icon| any]: On␣
↪your opponent's next turn you may prevent one spell of that type from taking affect.,␣
↪These walls have held off so many invaders\, I swear you can hear them say no if you␣
↪get close.
Bees, air, bees.jpg, (air, air), [], Gain 3\, 1/1 Air element Bee tokens, Not the Bees!
}
```

That's a lot of columns! We don't have to look at every one, but a couple stand outs:

- `cost` is a list where each value is an icon name
- `type` can be marked empty with `[]`
- we have a macro, `[text-icon| ]` to put costs into the effect

1.4.2 Special Sections

```
layout {
    template: euro-poker-deck.cls
}

macros {
    icon = <img source="img/[1]-small.png"/>
    text-icon = <img source="img/[1]-small.png" width="37" height="37"/>
}

defaults {
```

(continues on next page)

(continued from previous page)

```
    font-family: Cambria
    align: left, top
}
```

The only thing we define in `layout` is a template. This particular template is from the templates for The Game Crafter, their Euro Poker Deck is the same size as MtG cards. `macros` defines two functions, both turn a cost icon name into the actual image as an image tag for text, one of which defines a small size for text. Lastly `defaults` defines some features for all the text we have.

1.4.3 The Frame

```
bleed-zone {
    frame {
        type: image
        source: img/[frame]-bg.png
    }
}
```

The frame refers to the colored border around the card. In this case, where The Game Crafter expects full bleed, we use the template's `bleed-zone` element to position it. Remember, elements defined in a template keep all their properties unless you change any of them. The frame in this case also includes the boarders around the card title and text, just because it's easier to place them that way.

1.4.4 Title and Cost

For reference, everything else is a child of the `safe-zone` element of the template.

```
top-bar {
    size: 100%, .25in
    title {
        type: text
        text: [title]
        x: .1in
        font-size: .2in
    }
    cost {
        type: text
        text: [for-each| [cost], [icon| [item]]\s\s]
        align: right, middle
    }
}
```

Really the only thing that might raise an eyebrow here is the `cost` element. We use a `text` element because it's an easy way to hold an arbitrary number of images, and we can adjust the alignment as needed. We could even insert newlines in between the images to make a vertical cost. We also see a use of the `[for-each| ]` function and it's helper the `[item]` variable that returns the current value of the list.

1.4.5 The Art

```
img {
    type: image
    source: art/[art]
    position: 0.05in, .25in
}
```

Nothing to see here, move along.

1.4.6 Type and Card Text

```
type {
    type: text
    position: 0, 2in
    size: 100%, .25in
    font-size: 10pt
    align: center, top
    text: Spell [if| [ne| [type], []], - [type]]

}
card-text {
    position: .1in, 2.15in
    size: 2.13in, 1in

    effect {

        type: text
        size: 100%, 100%
        font-size: 9pt
        text: [effect]
    }
    flavor {
        type: text
        size: 100%, 100%
        font-size: 8pt
        align: left, bottom
        text: [i| [flavor]]
    }
}
```

Because these are all spells of various types, we know to put "Spell" in `type`, however if there's a subtype we also want a hyphen and that subtype. `effect` and `flavor` take up the same space, and are separated with their alignment, much like the top bar above.

1.4.7 Wrapping Up

That's everything! You can see how little you need to make a real card, it just takes time to find the ideal position for everything. What I recommend is to design the basic layout is something that reports real world units, like Inkscape, then translate that to CLS, that's what I did for this, although you'll still need to fine tune some numbers.

1.5 Quick Ref

This document is meant to show all the syntax of CLS in a smaller, yet more complete way than the tutorial. It's also easier to read than the full Syntax Reference.

1.5.1 Sections

Sections are made up of a name with contents surrounded by curly braces, like this

```
name {
    #contents go here
}
```

Generally contents are properties. Properties are a name, a colon, and a value

```
type: text
```

The value ends either at the end of the line or with a semicolon.

```
x: center; width: 1/2in
```

The whitespace (spaces and tabs) around names and values is ignored, so that width will be "1/2in" not " 1/2in". Whitespace within values is kept however.

Some properties take multiple values, that are separated by commas.

```
size: 1in, 1in
```

The `macros` and `data` sections have different contents. The `macros` section uses definition syntax, which is just property syntax using equals instead of colons. See the *macros section page* for more information. The `data` section uses CSV format, with either standard, excel-style syntax or CLS syntax which uses escapes to put commas in values, and ignores headers with no name.

```
macros {
    this-thing = some value I guess??
}
data {
foo, bar
[this-thing], I don't even know\, do you?
}
```

The `export` section and element sections can also feature subsections

```
export {
    pdf {
        name: more-cards.pdf
```

(continues on next page)

(continued from previous page)

```
    }
}

stats {
    attack {
    }
    defense {
    }
}
```

1.5.2 Elements

There are 6 types of elements

- `text` elements draw text both, simple and complex, and have access to a limited amount of HTML
- `image` elements draw images quickly and easily, best when one image will be used on every card
- `image-box` elements draw images according to an alignment, best when you have lots of images of different sizes
- `rect` elements draw rectangles, this is great for prototyping or drawing a bounding box to see what a given size would be
- `circ` elements draw circles
- `line` elements draw lines

Elements can also have subelements, which are drawn relative to their parent

```
container {
    subelement {
        type: text
        text: I'm drawn from the upper left of my container!
    }
}
```

1.5.3 Macros

Macros are either just a name surrounded by square brackets, or a name followed by a vertical bar for a function macro, with arguments separated by commas.

```
[column name]
[substr| [next column], 1, 1]
```

Most macros you'll use will likely be column or user variables. Macros are evaluated recursively, if a macro returns a macro then that macro will be evaluated too.

1.5.4 Comments

Comments must be on their own line and begin with a pound sign

```
# this is a comment
```

Comments can appear in any section.

1.5.5 Values

A value is anything that can be assigned to a property or given to a macro as an argument. Type of values are numbers, toggles, colors, and strings.

Numbers

Numbers take three basic forms, a whole number, a decimal, or a fraction

```
45
2
.3
1.45
3/4
1 1/8
```

Depending on the property or macro, numbers can have signs and units

```
-45
1 1/3in
```

Toggles

A toggle is a true or false value. There are three values for each true and false, the idea being some read better than others with different properties.

```
true
yes
on

false
no
off
```

Colors

A color is one of

- a hex code, either 6 digit #RRGGBB for opaque colors, or ##AARRGGBB for transparent colors
- a named SVG color name, like `blue` or `yellow`
- the color `transparent`

Lists

Lists are used to treat multiple values as a single value, mostly by macros. Lists are surrounded by parentheses and the items are separated by commas.

```
(red, blue, yellow)
(single item)
()
#an empty list
```

Strings

Strings are everything else. The fixed values for `align` and `decoration`, the path for `source` and the value for `text` are all strings. Strings also allow escapes, which is using a back slash before a character to show it isn't part of the syntax

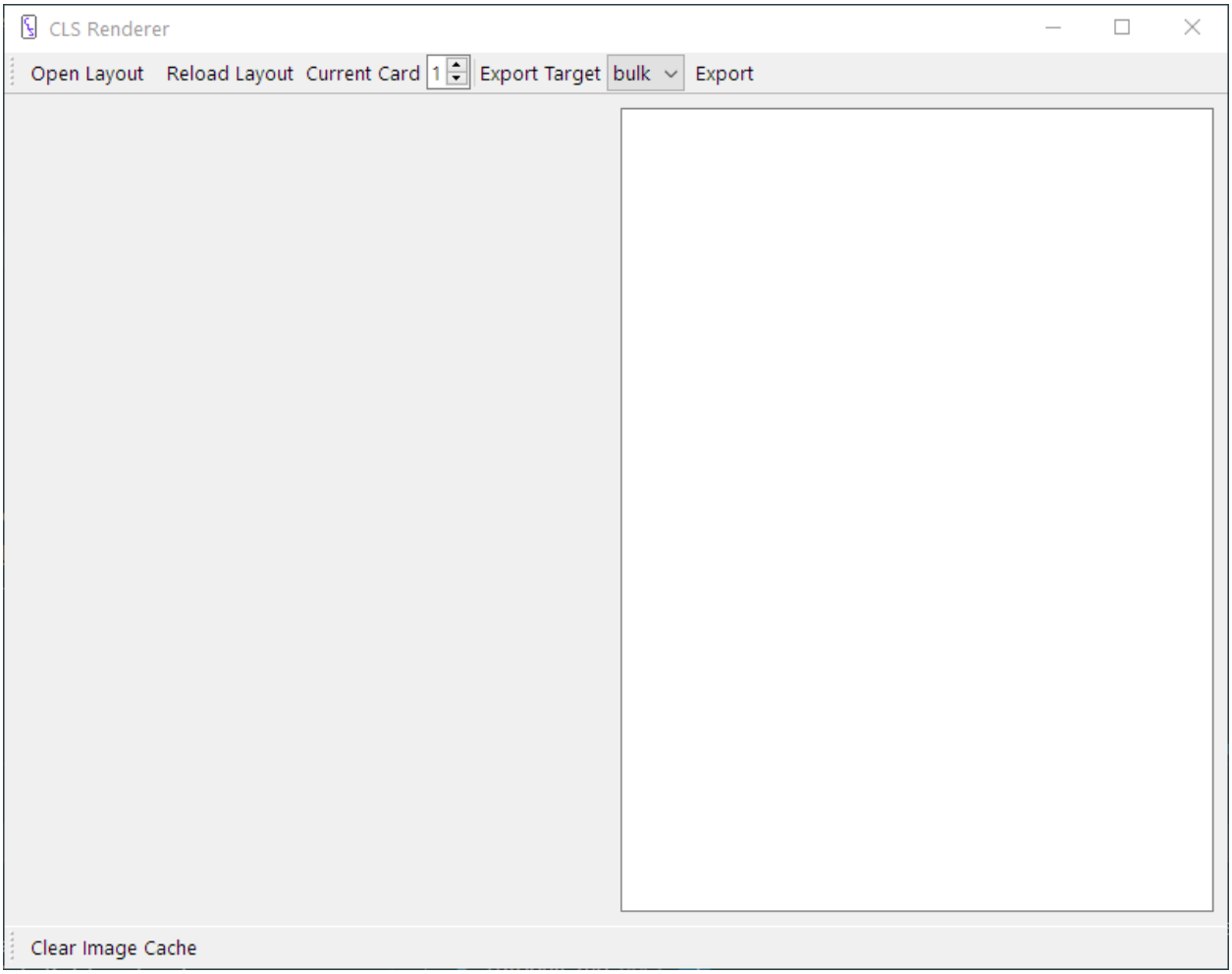
```
text: This is good!\[not it's not -ED\]
```

Four escapes have special meaning

Escape	Meaning	Escape	Meaning
`\n`	a new line	`\s`	a space
`\t`	a tab	`\\`	a literal backslash

The space and tab escapes are useful for putting whitespace at the edge of a value, where it would otherwise be stripped out by the CLS parser

1.5.6 Using the Renderer



The open button opens a new layout file and generates it.

The reload button reloads and regenerates the last loaded layout.

The current card spinner shows and changes the currently shown card.

The export target combo box selects which export target you want.

The export button exports the cards according to the export target.

The clear cache button on the bottom clears the image cache. The CLS Renderer stores any images that are loaded in to keep impact of the file system low. This button clears that storage so that if you change any images in the file system those changes will be seen next time you regenerate the cards.

The card is shown on the left, and on the right is a text box for displaying errors. Errors should be readable but if you have trouble trying to decipher them please ask on the CLS forum, located from the homepage.

1.6 Special Sections

The special sections control how the CLS Renderer operates, these are:

- `layout` - which holds the settings for the overall layout of the cards and settings for how to interpret values.
- `macros` - which holds user defined macros.
- `defaults` - which holds default values for element properties.
- `data` - which holds the data table used to generate cards.

1.6.1 The `layout` Section

The `layout` section follows the standard property-value format.

- `size: WIDTH, HEIGHT` - size of the card. Default value is `1in` for both values. Unit can be one of `px`, `in`, or `mm`.
- `bleed: BLEED-WIDTH(, BLEED-HEIGHT)` - bleed area for the card. Bleed refers to the area that is printed but not seen in a final copy. Unit can be one of `px`, `in`, or `mm`. Default value is `0, 0`
- `dpi: DPI` - dots (pixels) per inch used when converting inches and millimeters to pixels. Default value is `300`. No units are used for this number.
- `template: LAYOUT-PATH` - another layout to load in as a template. All properties and sections from the template will be overwritten by this layout. Any elements in the template will be drawn first before any elements defined by this layout.
- `data: FILE-PATH` - external data table to use, instead of the `data` section. Default value is `[]` which uses the `data` section.
- `csv: DIALECT` - dialect of csv to use for parsing the data table. Default value is `CLS`. Can be one of:
  - `CLS` - the native dialect used by CLS, as described in *Syntax*, which works well for hand written data.
  - `excel` - the dialect exported by spread sheet programs like Microsoft Excel and Google Sheets. This is a strict dialect and only recommended when using CSV data exported by another program.

1.6.2 The `macros` Section

The `macros` section uses a unique syntax to define user macros, which return their defined value. User macros can either be variables or functions. For example:

```
macros {
    blueish-gray = #8080aa
    the = THE [lower| [1]]
}
```

Which can be used as

```
some-element {
    ...
    text: [the| [card-name]]
    font-color: [blueish-gray]

}
```

When making functions the arguments passed to the macro are reachable by number as the macros `[1]`, `[2]`, `[3]` and so on. Also available to a function ares:

- `[arg-total]` - the number of arguments passed to the macro
- `[args]` - all the arguments passed to the macro as a list

1.6.3 The `defaults` Section

The defaults section defines default values for element properties, and as such looks like an element section with property: value format. The `defaults` section can feature any property an element can, such as `size`, `text`, or `scale` except `type`. When using shortcut properties in defaults, individual values can be changed in later elements, for example:

```
defaults {
    line: 0.1in, magenta
}
...
box {
    type: rect
    line-width: 50px
    line-style: dots
}
```

The element named `box` will have a line that's 50px wide, dotted, and magenta.

1.6.4 The `data` Section

The `data` section contains the data table, and uses CSV syntax. An alternate data table can be provided with the `data` property of the `layout` section. The dialect used to parse the data table is defined by the `csv` property of the `layout` section.

1.7 Export Section

Exporting cards produced by CLS Renderer is controlled by the `export` section and its subsections, each one controlling a different export target. These targets are:

- `bulk` - this target exports cards to individual images
- `pdf` - this target exports the cards to a pdf, useful print and play
- `tts` - this target exports the cards into an image suitable for Tabletop Simulator

The `export` section and subsections use property: value format. These properties are common to all export targets:

- `destination: FOLDER-PATH` - destination folder for exporting cards. Default value is `[]` which uses the same folder as the layout folder.
- `name: NAME` - the name to use when generating the files. Each export target handles the name differently due to each target saving different kinds of files.
- `include-bleed: TOGGLE` - whether to include the bleed area when exporting cards. Default value is `yes` for `bulk` export targets and `no` for all others.

Additionally the `export` section acts like the `defaults` section, but for export targets

```
export {
    name: cards
    bulk {
        name: card[card-index].png
    }
}
```

In this example the `pdf` and `tts` export sections will export cards as `cards.pdf` and `cards.png` while `bulk` will number each card.

1.7.1 The `bulk` Export Target

The `bulk` subsection contains properties for exporting cards to individual images.

- `name: NAME` - the name to use when saving the individual images. If no macros are present in `NAME` the variable `[card-index]` will be added to the front. Default value is `[card-index]card.png`.

1.7.2 The `pdf` Export Target

The `pdf` subsection controls the exporting process for PDF files.

- `name: NAME` - the name to save the PDF as. The file extension will always be converted to `.pdf`. Default value is `[]` which gives the pdf the same name as the layout file.
- `margin: X-MARGIN(, Y-MARGIN)` - the page margin. Cards will not be put in the margin. If `Y-MARGIN` is left off `X-MARGIN` will be used for both. Unit is one of `in` or `mm`. Unit for both must be the same. Default value for both is `.25in`, most printers lose accuracy when closer than this to the edge of the page.
- `border: WIDTH` - the width to draw a border around the cards with. If `0` no border will be drawn. A value of `1mm` will be appear on screen but not when printed. Unit is one of `in` or `mm`. Default value is `0.01in`
- `page-size: SIZE` the size of the page. Default value is `letter`. Must be one of:
  - `letter` - which is 8 1/2 by 11 inches or
  - `A4` - which is 297 by 210 millimeters
- `orientation: ORIENTATION` - the rotation of the page, either `portrait` or `landscape`. The default value is `portrait`.

1.7.3 The `tts` Export Target

The `tts` subsection is a specialized export target for Tabletop Simulator, which expects cards laid out in a grid. Tabletop Simulator also puts specific limits on the size of the image, namely that it must be 2-10 cards wide, 2-7 cards tall, and at max 4096 pixels wide. These are enforced by the CLS Renderer.

- `name: NAME` - the name to save the image as. If no file extension is present the extension `.png` will be added. Default value is `[]` which gives the image the same name as the layout file.
- `size: WIDTH, HEIGHT` - size, in cards, of the generated image. `WIDTH` must be between 2-10 inclusive and `HEIGHT` must be between 2-7 inclusive. Default is `5, 7`; a `WIDTH` of `5` is the max a US poker sized card at 300 dpi can have and keep under the 4096 pixel limit.

If there are more cards than can fit in a single generated image, the CLS Renderer will generate multiple images with a number appended to the front, eg `1cards.png` and `2cards.png`.

1.8 Elements and Properties

1.8.1 Elements

Elements are the building blocks of a layout. Anything that is seen on a card is an element. These are the available types of elements.

- `text` - This element renders text onto a card. The font family and color can be changed, text can be colorized and decorated (eg underlined), and structured with a subset of HTML
- `image` - This element is used to draw an image. Images can be resized while maintaining aspect ratio.
- `svg` - This element is used to render an SVG, either a whole document or an item defined by id.
- shapes - there are multiple elements for drawing basic shapes.
  - `rect` - a rectangle, with or without rounded corners.
  - `circle` and `ellipse` - both names are used for the same element. A basic circle or ellipse.
  - `line` - a line.
- `none` - this element has nothing to draw, but can contain other elements, have a position, and rotate.

1.8.2 Composite Properties

CLS makes use of two types of properties, single properties which take a single value, and composite properties which take multiple values separated by commas, for example:

- `keep-ratio: yes` is a single property
- `font: 1/4in, Impact, red` is a composite property

Composite properties are all made up of single properties, `font` above is made up of `font-size`, `font-family`, and `font-color`. Not all single properties are a part of a composite property, `keep-ratio` for example is only available alone. As a guideline, composite properties are recommended over single properties when available, but sometimes single properties offer better readability when using macros.

Some composite properties will be listed after their separate single properties when the values are different types. When the values are the same type, the composite property will be listed alone and merely state it's a composite property. In these cases the single properties are named for the value names shown in the description, so `size: X, Y` can be broken down into `x` and `y`.

A small number of composite properties can take optional values, such as `scale: SCALE-WIDTH(, SCALE-HEIGHT)`. In these cases the optional value is enclosed in parentheses, and the behavior if omitted will be described in the property listing.

1.8.3 Common Properties

All elements share these properties.

- `type: TYPE` - the type of the element, as listed above. This must be a literal value like the values used in the special sections; this means that macros will not be evaluated. Default value is `none`.
- `position: X, Y` - the position of the element. Default value for both is `0`. Composite property. Allowed values are any of
  - A number with units `px`, `in`, or `mm`, either positive or negative. This positions the element's upper left to its container's upper left. This is considered the standard positioning scheme.

  - A number with units `px`, `in`, or `mm`, and with the sign `^`. This positions the element's lower right to its container's lower right. This and the above type can be combined, eg `position: 1/4in, ^1/4in` would position an element based on it's lower left.
  - A number with unit `%` which positions the element relative to its container in the same direction, eg `x: 50%` will position the left edge of the element in the middle of its container going left-to-right.
  - `center` which centers the element within its container.
- `size: WIDTH, HEIGHT` - the size of the element. Both `WIDTH` and `HEIGHT` must be present. Default value for both is `1/4in`. Composite property. Allowed values are any of
  - A number with units `px`, `in`, or `mm`.
  - A number with unit `%`, which sizes the element relative to its container.
- `angle: ANGLE` - the rotation of the element in degrees, with or without the unit `deg`. Can be positive which rotates clockwise or negative which rotates counter-clockwise. Elements rotate around their center. If a container is rotated, subelements are positioned according to that rotation. Default value is `0deg`.
- `draw: TOGGLE` - whether to draw this element, also affects any subelements. Default value is `yes`.

1.8.4 The `text` Element

- `text: TEXT` - the text to display. Default value is `[]`.
- `font-size: FONT-SIZE` - the size of the text. Default value is `18pt`. Unit is one of `pt`, `px`, `in`, `mm`.
- `font-family: FONT-FAMILY` - the font family to use. Default value is `Verdana`.
- `font-color: FONT-COLOR` - the color of the text. Default value is `black`.
- `font: FONT-SIZE, FONT-FAMILY(, FONT-COLOR)` - this is the composite property for the above three properties.
- `font-weight: WEIGHT` - the heaviness of the font, eg `400`, or `100`. Used for fonts with multiple weights. If given the renderer will ignore the `bold` property or a `bold` value given to `decoration`. Default value is `[]`.
- `shrink-font: LENGTH-FONT-SIZE-PAIRS` - a list of pairs that match a length for `text` to a new font size. Pairs are split by `:` colon. `LENGTH` is a number without a unit and `FONT-SIZE` uses the same units as the `font-size` property above. Default value is `()`. See the example below for more information.
- `h-align: H-ALIGN` - the horizontal alignment of text within the element. Default value is `center`. Allowed values are:
  - `left`
  - `center`
  - `right`
  - `justify`
- `v-align: V-ALIGN` - the vertical alignment of text within the element. Default value is `top`, allowed values are:
  - `top`
  - `middle`
  - `bottom`
- `align: V-ALIGN, H-ALIGN` - this is the composite property for the above two single properties.

- `decoration: DECOS` - decoration options for the text, separated by commas. Default value is `[]`. Values are allowed in any order, and each `DECO` has it's own single property that takes a toggle, allowed values are:
  - `italic`
  - `bold`
  - `overline`
  - `underline`
  - `line-thru` or `line-through`

HTML subset

Because CLS Renderer uses Qt for rendering cards, a subset of HTML is provided for `text` elements, such as:

- bold text, also usable thru the `[b| ]` macro which wraps it's argument in   tags
- italic text, also usable thru the `[i| ]` macro which wraps it's argument in   tags
- <font> allows you to change the font family in the middle of a label
- <img> places images in the text. When used in conjunction with user macros this is a convenient way to add icons to text

These are only some of the tags available, for a fuller explanation of these and more valid HTML, visit the Qt docs for the Supported HTML Subset

`shrink-font` example

The `shrink-font` property is used to create a "shrink to fit" effect. When the text is longer than a given length, the font size is changed to match. Take the example below for a card effect box from a trading card game.

```
font-size: 11pt
shrink-font: (200: 9pt, 300: 7.5pt)
text: [card effect column from data]
```

In this example, most cards will have a font size of 11pt, but if the length of an effect (minus HTML) is equal to or greater than 200 characters, the font size will be set to 9pt, and similarly with 300 characters and 7.5pt. So if the length of an effect is 256 characters, the font size would be 9pt. These sizes are check in the order they appear in.

1.8.5 The `image` Element

This type sets the default size to `0, 0`, the actual size the image is drawn is found using the method described below.

- `source: SOURCE-PATH` - the image file to load. Most common file types are recognized, but png is recommended because it's a lossless format. If the specified image isn't found nothing will be drawn. Default value is `[]`, no image.
- `keep-ratio: TOGGLE` - whether to keep the image's original aspect ratio when resizing the image. Default value is `yes`.
- `scale: SCALE-WIDTH(, SCALE-HEIGHT)` - amount to scale image by. Unit is one of `x`, `%`, or `dpi`. Composite property. Default value is `1x` which means no scaling takes place. If `SCALE-HEIGHT` is left off `SCALE-WIDTH` is used for both directions.
  - Values with unit `x` are factors, where `1x` is full size, `.5x` is half size, `2x` is double and so on.

  - Values with unit `%` are percentages, where `100%` is full size, `50%` is half size, `200%` is double and so on.
  - Values with unit `dpi` are dots per inch. Use this unit to specify that an image was drawn at a different dpi than the card is drawn at. For example, the default card dpi is 300, but artists will commonly draw at 600, so specifying `600dpi` will cause the image to be drawn so it takes up the same real world space.

The final size to render an image is found with the series of checks below.

1. If the size and scale are unspecified, the image is drawn at full size. This is the default behavior.
2. If the size is unspecified but `scale` is provided, the image is scaled based on the provided value and according to `keep-ratio` when drawn.
3. If the size is specified in only one direction the image is scaled based on that direction while maintaining the aspect ratio when drawn.
4. If both width and height are specified, the image is scaled to that size according to `keep-ratio` when drawn.

1.8.6 The svg Element

Everything said about `image` above is true for `svg`, however it adds one more property.

- `id: ID` - an id of a specific element in the SVG document. Default value is `[]`, which draws the entire document.

1.8.7 The Shape Elements

CLS provides basic shapes. While mainly intended to assist in development of layouts, these may also be used, for example, to provide borders in more simple layouts, or to colorize transparent images.

All shapes share the following properties:

- `line-width: LINE-WIDTH` - the width of the line used to draw the shape. Unit is one of `px`, `in`, `mm`. Default value is `0.01in`.
- `line-color: LINE-COLOR` - the color of the line. Default is `black`.
- `line-style: LINE-STYLE` - the style of the line. Default is `solid`. Allowed values are:
  - `solid`
  - `dots`
  - `dash`
- `line: LINE-WIDTH, LINE-COLOR(, LINE-STYLE)` - this is the composite property for the above 3 properties.
- `fill-color: FILL-COLOR` - the color on the inside of a shape. Default value is `white`.

The `rect` type has one additional property:

- `corner-radius: X-CORNER-RADIUS(, Y-CORNER-RADIUS)` - this property controls the rounding of the corners of a rectangle. Unit is one of `px`, `in`, or `mm`. Composite property. Default value is `0, 0` which is no rounding. If `Y-CORNER-RADIUS` is left off `X-CORNER-RADIUS` is used for both directions.

The `circle` and `ellipse` types have one additional property:

- `diameter: WIDTH(, HEIGHT)` - this property is the same as `size` except that if `HEIGHT` is not provided `WIDTH` will be used for both dimensions.

The `line` type cannot be rotated or given a size, and has two additional properties:

- `start: X, Y` - this property is the same as `position`.

- `end: X2, Y2` - this property defines the end point of the line. Allowed values are the same as for `position`. Composite property. Default value is `1/4in, 1/4in`, which is also the default for `size`.

1.9 Macros

Macros are bits of text expanded by the CLS Renderer to provide dynamic, per card, text. Macros come in three varieties:

- variables like `[card-index]` that simply return a value
- functions like `[if| ]` that process arguments, and
- operator macros that perform special parsing for operators on their argument

1.9.1 Variables

The variable macros you'll most likely see are column variables, macros that are filled in by the renderer based on the current data row, and user variables, the variables defined in the `macros` section. Here are some others.

`[]` - the empty macro Use this macro when you don't want to fill in a value or argument and want to make it clear that the blank space is intentional.

`[row-index]` and `[row-total]` `[card-index]` and `[card-total]` `[repeat-index]` and `[repeat-total]` These variables return numerical statistics about cards

- `row` refers to the current row of the data. Cards generated from the first row of data will have a `[row-index]` of `1`
- `card` refers to the specific card. The total number of generated cards is in `[card-total]`
- `repeat` refers to the repetitions as defined by a repeat column in the data. `[repeat-total]` is the same as the repeat value of a given row. If no repeat column is present in the data `[repeat-index]` and `[repeat-total]` will both be `1`
- `index` refers to the number of the current card. The second card made from a row will have a `[repeat-index]` of `2`
- `total` refers to the total number of that category. `[row-total]` is the total number of rows

These are useful for things like generating "24 out of 50" with `[row-index] out of [row-total]` where repeated cards are not counted separately. If the layout is being generated without data, these will have an undefined value, they could be one, they could be blank.

1.9.2 Functions

Functions can effectively be divided into two categories, value functions and comparison functions.

Value Functions

Value functions modify and create values.

`[b| STRING]`
This function is a shortcut for bolding text with STRING in labels.

`[capitalize| STRING]`
Capitalize `STRING`. This uses a dumb algorithm of making the first letter of any word longer than 4 letters, plus the first letter overall, uppercase.

`[dup| TIMES, STRING]`
Duplicate `STRING` `TIMES` times. When evaluating `STRING`, the variable `[d]` is set to which duplication this is, eg `[dup| 3, \s[d] times]` will return " 1 times 2 times 3 times". If `TIMES` begins with a `0` then `[d]` will start counting with zero.

`[file| FILENAME]`
Open the file `FILENAME` and return the text.

`[for-each| LIST, STRING]`
Evaluate `STRING` once for each item in list. When evaluating `STRING`, the variable `[item]` is set to each item in `LIST` in order. This is useful when paired with the `[args]` variable available to user functions, eg `[for-each| [args], [item] &\s]`

`[i| STRING]`
This function is a shortcut for italicizing text with STRING in labels.

`[lower| STRING]`
Convert the entirety of `STRING` to lowercase.

`[rnd| STOP]`
`[rnd| START, STOP]`
Generate a random number from `START` up to and including `STOP`. `START` must be a lower number than `STOP`.

`[s| STRING]`
This function is a shortcut for striking thru text with <s>STRING</s> in labels.

`[slice| STRING, START]`
`[slice| STRING, START, END]`
Select a sub string from `STRING` starting at `START` and ending with `END`. If `END` is not present then the rest of the string will be selected, if it is present the specified character won't be included. If `START` or `END` begins with a `0` that argument will count the first character as `0` the second as `1` and so on, otherwise the first character is `1`. Negative numbers are also allowed, which are counted from the end of the string, so `-1` is the last character.
`[slice| LIST, START]`
`[slice| LIST, START, END]`
Select a sub list from `LIST`. The `START` and `END` arguments are as above, where `START` is the first item to return.

`[substr| STRING, START, LENGTH]`
Select a sub string of `STRING`, starting at `START` for `LENGTH`. If `START` begins with a `0` the first character will count as `0` the second as `1` and so on, other wise the first character is `1`. Negative numbers are not allowed.

Examples

Because `[slice| ]` and `[substr| ]` are so similar and so flexible, examples for both are located on this page.

`[u| STRING]`
This function is a shortcut for underlining text with <u>STRING</u> in labels.

`[upper| STRING]`
Convert the entirety of `STRING` to uppercase.

Comparison Functions

Comparison functions compare values and return a toggle for the `[if| ]` macro, which is also described here

`[either| LEFT, RIGHT]`
Returns `LEFT` if it is a true value, such as `on` or any string, or `RIGHT` if `LEFT` is false, such as `[]`. This function can be used in user functions to provide default values, such as `[either| [1], some value]`, where if the user doesn't provide a value, `some value` will be used instead.

`[eq| LEFT, RIGHT]`
Test if `LEFT` and `RIGHT` are equal. The arguments can be any type of value, but are compared as strings so `[eq| 3in, 3]` would return `false`.

`[if| TEST, TRUE]`
`[if| TEST, TRUE, FALSE]`
Returns either `TRUE` or `FALSE` depending on `TEST`, which must evaluate to a toggle. When the first character of `TEST` is `?` the rest of the argument will be given to the comparison macro for evaluation. The `FALSE` argument is optional, and omitting it is the same as using `[]` as `FALSE`.

`[in| VALUE, ARGS...]`
This function takes any number of arguments. Test to see if `VALUE` is equal to any of `ARGS`.
`[in| VALUE, LIST]`
Test to see if `VALUE` is in `LIST`. If more arguments are provided this macro will operate as above.

`[ne| LEFT, RIGHT]`
Test if `LEFT` and `RIGHT` are not equal. The arguments can be any type of value, but are compared as strings so `[ne| 3in, 3]` would return `true`.

`[not| TOGGLE]`
Flips `TOGGLE`, that is, if `TOGGLE` is a false value, like `off` or `[]`, then it returns `true`, otherwise returns `false`.

`[switch| TEST, CASE, MATCH...]`
`[switch| TEST, CASE, MATCH..., default, DEFAULT-MATCH]`
This function takes any number of `CASE, MATCH` pairs. Test to see if `TEST` is equal to any `CASE` and return the corresponding `MATCH`. If `CASE` is a list then this this function checks to see if `TEST` is in `CASE`. If the last case is `default` and `TEST` does not match any `CASE` value, then `DEFAULT-MATCH` will be returned.

Say you're using `[repeat-index]` to make numbered cards but you want some numbers to be drawn differently

```
[switch| [repeat-index], 10, X, (6, 9), [u|[repeat-index]], default, [repeat-index]]
```

This would return `A` if the value is `1`, underline the value if it's `6` or `9`, or else return the unchanged `[repeat-index]`.

1.9.3 Operator Macros

Operator macros scan their argument for operators, special characters that the macro gives special meaning.

`[?| VALUE ]` - the comparison macro
The comparison macro performs numeric comparison. `VALUE` is evaluated for a single comparison operator with either side being the operands, if either operand is not a number parsing will stop with an error. Units are ignored so `3in`, `3mm`, and `3%` are all treated the same as `3`. Valid comparison operators are:

- == - equal to
- != - not equal to

- > - greater than
- >= - greater than or equal to
- < - less than
- <= - less than or equal to

When the first character of `TEST` of an `[if| ]` is `?` the rest of the argument will be given to this macro for evaluation.

```
[if|? [card-index] == [card-total], Last card!, Still waiting...]
```

This would evaluate to "Last card!" on the last card and "Still waiting…" on every other card.

`[=| VALUE ]` - the math macro
The math macro performs arithmetic. `VALUE` can contain any number of operators and they will be processed according to order of operations. If any operand is not a number parsing will stop with an error. Units are ignored like the comparison macro above. Operators and numbers must be separated with spaces, as in `1 + 2` but not `1+2`. Accepted operators are:

- `+` - addition
- `-` - subtraction
- `*` - multiplication
- `/` - division
- `%` - modulus, the remainder of division
- `(` and `)` - grouping

To provide an example:

```
[=| [card-index] / [card-total] * 100]
```

This would give `[card-index]` as a percent, for example the 21st card of 34 would be "61.76". We can use this to draw a progress bar as in

```
bar-holder {
    width: 2in
    ...
    bar {
        type: line
        ...
        x: 0
        x2: [=| [card-index] / [card-total] * 100]%
    }
}
```

The element `bar-holder` holds how long the bar can grow to, and `bar` is the actual progress bar. The same example above would get us a bar that's about an inch and a quarter long.

1.10 Glossary

The CLS Renderer has many moving parts and it's important to keep consistent names for things to not get too confused.

Argument
: a value passed to a function macro.

Card
: The final rendered object made by the generator, does not need to be an actual card. This could also be a token or board, a character sheet, or something not from the table top game space like a letter or mailer, or a sprite used by a video game.

Card Layout Script
: The name of the programming language used by the CLS Renderer

CLS Renderer
: The app that turns a layout file into cards

Container
: An element that contains another element, or the layout if a given element has no container. The phrase "element container" is used to specify the first case.

Compiling
: The process of turning elements as described by the user into object usable by the renderer

Data
: The CSV based based data that is used to fill in and customize specific elements of a layout.

Element
: A portion of a Layout, for example an image.

Expand
: To process a value into one that contains no macros or escapes.

Function
: A macro that takes arguments. These always compute a return value.

Layout
: A template for making Assets. A layout contains data about itself, as well as a number of elements.

Layout File
: A file containing a layout written in CLS.

Macro
: A piece of text like `[asset-index]` that returns another piece of text. These are like the variables and functions of programming languages.

Property
: A changeable feature of a layout or element, for example `width` or `angle`.

Rendering
: The process of drawing elements onto a card.

Value
: The assigned contents of a property. In `width: 1in` the `1in` is a value.

  The term value is also used when referring to the data, where it refers to the specific values available as column macros.

Variable
: A macro that does not take arguments. These generally do not compute their return value.

1.11 FAQs

Okay, so realistically most of these have been asked once at the most, but "Potentially Asked Questions" doesn't sound as good.

1.11.1 So what is CLS?

It's a programming language used to describe card layouts. The renderer combines layouts with data in CSV format to make cards. The driving goal is to make a simple to use, easy to read language that make the simple stuff easy and the hard stuff possible.

1.11.2 Is there a graphical editor?

There is not currently, no. I have very little experience making graphical applications and the time to make CLS would double. Additionally at this point so many features, particularly macros, are wrapped up in the text based format, that the longer I work on CLS the less compatible with a graphical interface it becomes.

If someone else wants to work on a separate graphical editor I'd be happy to work with them to make that happen.

1.11.3 Why should I use CLS instead of (some other card designer)?

If you're in the middle of a project and you're happy with what you're using, there's no reason to change. But try out CLS for your next idea, see how it works for you.

CLS is still young and fairly untested. Most other card designers will have more features and be less likely to have bugs, but I'm actively working on making CLS stable and featureful.

1.11.4 I think I found a bug!

Very possible! The best way to tell me is to make a post on the forum located on the home page, but if you don't want to make an account for that, you can email me at codlark (at) gmail (dot) com with "CLS BUG" in the subject and I'll double check the bug and make the post for you.

1.11.5 Are you going to add (some feature)?

I've likely thought about it and either have it in my personal planning doc or have ruled it out for some reason. Make a post on the forum located on the home page and I'll get back to you on that. For this I do ask that you post on the forum so that it's easier to have a back and forth about the feature if need be.

1.12 Syntax

This page is meant as a thorough, technical explanation of the syntax, parsing, and card generation of CLS. It is not meant to be a learning tool, rather a reference in the event that something doesn't act as expected.

1.12.1 Top Level Syntax

The 'top level' refers to the first layer of sections in a layout, where the special sections are defined.

Sections are a name followed by contents contained within curly braces.

```
layout {

}
```

Section names can contain white space, but it's not recommended. This form is a 'section definition'. The top level of the file can only contain section definitions.

1.12.2 Whitespace

Whitespace refers to non visible characters, in the context of CLS this means spaces and tabs. White space is ignored on boundaries, but left intact within names and values. in the following example, ignored whitespace is left blank, while non-ignored whitespace is marked with a dot •.

```
macros {
    my•name = my•name•is•[ some•column ]
}
this•section { draw : no }
```

1.12.3 Property Syntax

All sections except for the `macros` section and the `data` section use 'property syntax'. Properties are a name followed by a colon, text, then a terminus, which is one of a new line, a semicolon, or the end of a section.

```
image {
    scale: 50%; keep-ratio: yes
    source: img.png}
```

Some properties are 'composite properties' and some are 'single properties'. Single properties only take a single value and the entirety of text between the colon and the terminus becomes the value. Composite properties take multiple values. The text between the colon and the terminus is divided by commas into separate pieces of text when the property is parsed when the layout is first loaded. Attempting to use a single macro to fill multiple values will result in errors.

Properties do not need to be indented within a section, this is merely a convention to make layouts easier to read.

1.12.4 Subsections

A subsection is a section definition contained within another section. Only the `export` special section and element sections can contain subsections.

1.12.5 Definition Syntax

Definition syntax is only allowed within the `macros` special section, and is used to name and define macros. Macro definitions are a name followed by a an equals, text, then a terminus, much like property syntax.

```
macros {
    some-color = #343390
    another color = #769870
}
```

Names can contain whitespace, so both names above are valid

1.12.6 Data Syntax

The `data` special section uses a common format called comma separated value, or CSV for short. CSV is an old format and as such there are multiple dialects. CLS recognizes both it's own dialect optimized for writing data for layout files by hand, and the "excel" dialect used by spreadsheet applications that is meant for programmatic generation and consumption. This document will only describe the former.

The contents of the `data` special section is called the 'data'. Rows are split into rows on newline characters and there is no way to change this behavior. The top row of the data is a row of 'headers', these will become the names of column macros by which the text of the data can be reached. If there are any blank headers they will be ignored. If a row does not provide text for every column, the unspecified columns will be given blank text.

```
color left, color center,, color right
red, blue, yellow
black, , white
, purple
orange, green
```

This example creates three column macros, `[color left]`, `[color center]`, and `[color right]`, and the following four rows.

color left	color center	color right
"red"	"blue"	"yellow"
"black"	""	"white"
""	"purple	""
"orange"	"green"	""

Each row will generate a single card. And for each card, the column variables will resolve to their respective text.

1.12.7 Macro Syntax

Macros come in two varieties, variables that return text, and functions that take arguments to return a computed text. Macros are a name surrounded by square brackets. In function macros a vertical bar is used to separate the name from the arguments, and commas are used to separate the arguments from each other.

```
some-element {
    draw: [in| [repeat-index], 1, 2, 4, 6]
}
```

By convention, spaces are not put around names.

1.12.8 Comments

A comment is a line of text that is ignored by the parser. Comments must be on their own line and their first non whitespace character must be a pound.

```
#this is a comment
```

Comments can appear in any section.

1.12.9 Text and Values

Anything provided to a property is considered text until it gets processed at card generation time. Text is turned into a value, depending on how a given property processes this text, each time a card is generated. There are two kinds of values, generated values and literal values.

A generated value is any text that contains macros. This includes column variables. Any macros found are resolved into text. If the text contains macros after this, those macros are resolved. This is repeated until there are no more macros found in the text. The text is then treated as a literal value.

A literal value is text that does not contain macros. This includes special values such as the `center` used by the `position` property. properties belonging to special sections only take literal values, as does the `type` property of elements.

1.12.10 Number Syntax

Numbers in CLS are made up of:

- a sign
- an integer portion
- either
  - a decimal point
  - a decimal portion
- or
  - a space, decimal point, or slash
  - a fractional portion
- a unit

A sign is one of `+`, `-`, or `^`. If no sign is given the sign of `+` is used. Only some properties and macros allow negative numbers with the sign `-`. The sign `^` is only allowed with the `position` property.

The integer and decimal portions are made up of the digits `0` thru `9`; hexadecimal numbers are not allowed. The decimal point and decimal portion are not required for whole numbers. The integer portion is not needed for numbers between 1 and 0 or numbers between 0 and -1.

The fractional portion is made up of a `/` with digits on either side.

The allowed units are listed in the property description, with the first listed unit being the default unit used if the number does not have a unit.

Spaces are only allowed in numbers to separate an integer portion from a fractional portion. Examples of numbers include

```
0
1.2in
44%
-4.4
5
5.5
5 1/2
.5
1/2
```

When numbers are used as arguments to a macro, be it the math macro, the slice function, or any other, units are ignored, and may even be an error.

```
[if|? 1mm == 1in, will always be true, this is never seen on a card]
```

1.12.11 Toggle Syntax

A toggle is a value that is either true or false, on or off. Multiple values are allowed for each option to better suit the property. True values are

- `true`
- `yes`
- `on`

False values are:

- `false`
- `no`
- `off`

1.12.12 Color Syntax

Colors use the standard hex code format, a pound sign with 6 or 8 hexadecimal numbers after, in the order transparency, red, green, blue, such as `#45454545` or `#808000`. A collection of named colors are also available, with names taken from the SVG color keyword names with the addition of `transparent` which is a fully transparent color.

1.12.13 List Syntax

A list is multiple values taken as a single value. Lists are surrounded by `(` and `)` and list items are separated by `,`. If the last value in a list is blank then it isn't added to the list. If a blank value is needed as the last value, a `,` should be put after it.

```
(red, blue, yellow, green, black, white)
(1, 345, 2, 56, 7, 23, 4)
(a single item)
()
# empty list
([])
# also empty
```

(continues on next page)

(continued from previous page)

```
([],)
# this one contains a single blank value
```

1.12.14 Strings

Any value that is not a number, a toggle, or a color is a string, this includes file paths and fixed values such as `italic` or `dots`. String values are the only place escapes are processed. A blank string can be indicated with the blank macro `[]`.

Escapes

An escape is a sequence of characters used to indicate that some part of the escape is nor parsed as normal. CLS uses a common method of using the backslash to prevent the parser from seeing the next character.

```
text: I might\; or I might not.
```

In this example the first semicolon is escaped and the value for the `text` property is "I might; or I might not." Four characters are treated specially when they are escaped.

Escape	Meaning	Escape	Meaning
`\n`	a new line	`\s`	a space
`\t`	a tab	`\\`	a literal backslash

Every other character is left alone and passed along without transformation. This includes the semicolon above and the various separators and delimiters described elsewhere. Escapes are transformed after the whitespace is stripped from a value, so a single `\s` would create a value of " ". This can be used to put whitespace at the edge of a value.

In the CLS dialect of CSV, escape syntax can be used to include a comma in a value.

File Paths

A file path is a path to a file or folder in the file system. Properties that take paths have their value marked with `-PATH` in their description. Because CLS uses the same character as Windows uses as a file path separator, the forward slash is also allowed as a file path separator by CLS, and will be converted as appropriate.

```
source: images/backgrounds/[type].png
```

1.13 Rendering

Rendering is the process of turning a parsed layout into cards. If no data is present, only one card will be generated, and the index and total macros will all resolve to `1` or `[]`.

If there is data present, each row will generate at least one card. If the data contains a column named `repeat` that column must only contain positive numbers with no unit, and each row will generate as many cards as specified by the repeat value.

For each row, each element of the layout is processed. For each element, each property text is turned into a value, then that element is drawn to the card. Cards sit in a buffer until they are exported. Errors in parsing the file, turning text into values, or trying to use values to draw elements, will cause an error to be reported to the user.

other stuff

1.14 Changelog

This page collects the changes made to CLS Renderer over time

1.14.1 v1.3

added

- a new type, `svg`, allowing you to render SVG files, either the whole thing or an item specified by id
- a new scale unit for `image`, `dpi`. This unit was added for `svg` elements, but was added to `image` to maintain parity between the two types
- a new `font-weight` property has been added to `text` elements. This property takes a number and changes how heavy a font is. This is only needed for some more complete fonts that provide multiple weights like Thin, Light, Normal, Bold, Super Bold. Any value other than `[]` will override the `bold` property.
- a new `[not| ]` function that flips a toggle.
- a new `[either| ]` function that takes two values and returns the first one unless it's a false toggle or an empty string, in which case it returns the second value.

changed

- the default unit for the `scale` property of `image` is now `x`

fixed

- percent sizes of elements contained directly by the layout were being based on the full size of the card with bleed. This has been fixed and is now based on the size of the card as reported to the `size` property of the `layout` element

removed

- the `image-box` type. The provided functionality can be replicated with a container, and quite frankly having two images types is confusing.

1.14.2 v1.2

added

- A new `shrink-font` property has been added to `text` elements. This property takes a list of lengths and font sizes, and when the `text` property is longer than a given length, the font size is switched to the given size. See the docs for more info.

fixed

- apparently angle brackets weren't being interp'ed right! So I added proper escapes for them
- there was an inconsistency with using `pt` sizes with fonts. they should now scale properly based on your dpi at 72 points to an inch
- the docs said that bleed could take a size in `px` but this was raising an error. This no longer raises an error

1.14.3 v1.1

added

- added back `[substr| ]`, just because it makes more sense for some things.

changed

- Changed the licensing to the Mozilla Public License 2.0
- Syntax for macros has been changed. The separator between arguments is now the comma, the vertical bar is still used is separate the function name form the arguments, for example `[eq| [role], werewolf]` from the werewolf example.
- Syntax for lists has changed. Values are now separated by commas, as in `(red, red, blue)`
- lists can now be used with `[switch| ]`. When a list is used as a case value, the macro will check if the test is in the list. Check the docs for more.

1.14.4 v1.0

The program has been renamed to CLS Renderer, and briks have been renamed macros.

added

- `scale` property on `image` and `image-box` now takes factors with unit `x` in addition to percentages. `scale: 2x` is the same as `scale: 200%`
- `[switch| ]` macro. The first argument is a test, the rest of the arguments are put into case|result pairs. if test equals a case the result is returned.
- A new list value type that looks like `(a:b:c:d)` or `:` for an empty list
- New macros, and alterations to existing macros to make use of this new type
  - `[in| ]` will check if its second argument is a list, and if so will check if the test value is in that list
  - `[slice| ]` will operate on list items, and will return a list, when passed a list instead of a string

  - `[for-each| ]` evaluates a body for each item in a list
  - `[length| ]` returns the length of a list or string
  - `[args]` is available to user function macros to return all arguments passed to the macro as a list
- "Clear Image Cache" button to the interface.

changed

- the `briks` section has been renamed `macros` as the name brik has been replaced by macro
- the `output` property on export sections has been renamed to `destination`
- mixed fractions can have their whole number separated from fraction component with a space, a period, or a slash. These three are all valid: `1 1/2` `1.1/2` `1/1/2`

fixed

- parsing mixed fractions in the math macros now works as intended
- using the `[file| ]` macro with a file that doesn't exist/won't open should give a proper error now.
- using `^` when the parent is layout and bleed is non zero now works as intended
- there was a rounding error when calculating the full size of a card with bleed greater than 0, it's been fixed

removed

- The `[substr| ]` macro has been removed. Use slice instead

1.14.5 v 0.6

added

- composite properties, which take multiple values and pass them off to specific properties. Composite properties are also preferred when available
- `export` section with subsections that describe how to export to various targets
- `scale` property to `image` and `image-box` element types to control image size proportionately
- users can now make function briks. If a user brik uses the briks `[1]`, `[2]`, and so on they'll be filled in by the arguments passed to the brik

changed

- properties and other names previously in camelCase, like `fontSize` now use hyphen as in `font-size` and `asset-index`
- fractional numbers now use a space instead of a decimal point

removed

- `name` property on `layout` section
- dedicated `pdf` section, now merged with new `export` section

fixed

- rotating `image-box` elements is less surprising

note

- added in 0.5 but not mentioned, a `[rnd| ]` brik

1.14.6 v 0.5

- added: `imageBox` element type, which aligns images within its box
- added: `csv` property to layout that controls what dialect of csv to parse data as
- changed: unknown properties are no longer errors

1.14.7 v 0.4

- added: more units! checkout Values(page removed)) for how they work
- added: `[slice| STRING | START | STOP]` takes substring of `STRING` like `[substr| ]` but uses start and stop positions instead of a start position and a length
- added: support for parenthesis to the math brik, infact the whole thing has been rewritten
- added: pdf export
- changed: the `[substring| ]` brik has been renamed to `[substr| ]`
- changed: the math brik has been changed to `[=| ]` and has different semantics

1.14.8 v 0.3

- added: A graphical interface
- added: elements can now contain other elements
- added: `defaults` section
- added: `[s| ]` and `[u| ]` for strike thru and underline respectively
- changed: `names` section now called `briks`
- changed: new css style syntax
- changed: image resizing ruels have been tweaked
- changed: completed the playing card example and corresponding help page
- changed: tweaks to the csv dialect
- fixed: layouts without a `layout` section now emit a helpful error

- fixed: somehow the `[file| ]` brik wasn't registered properly, it's fixed now

1.14.9 v 0.2

- added: templates for layouts
- added: a new playing card example
- added: `draw` property on elements that controls if an element gets drawn
- added: math brik, `[#| ]`, that processes arithmetic
- added: `[file| ]` brik that loads in the contents of a file
- changed: syntax is now in a more modern colon-indent style
- changed: images are now cached when loaded for the first time
- fixed: single column data sets now work right