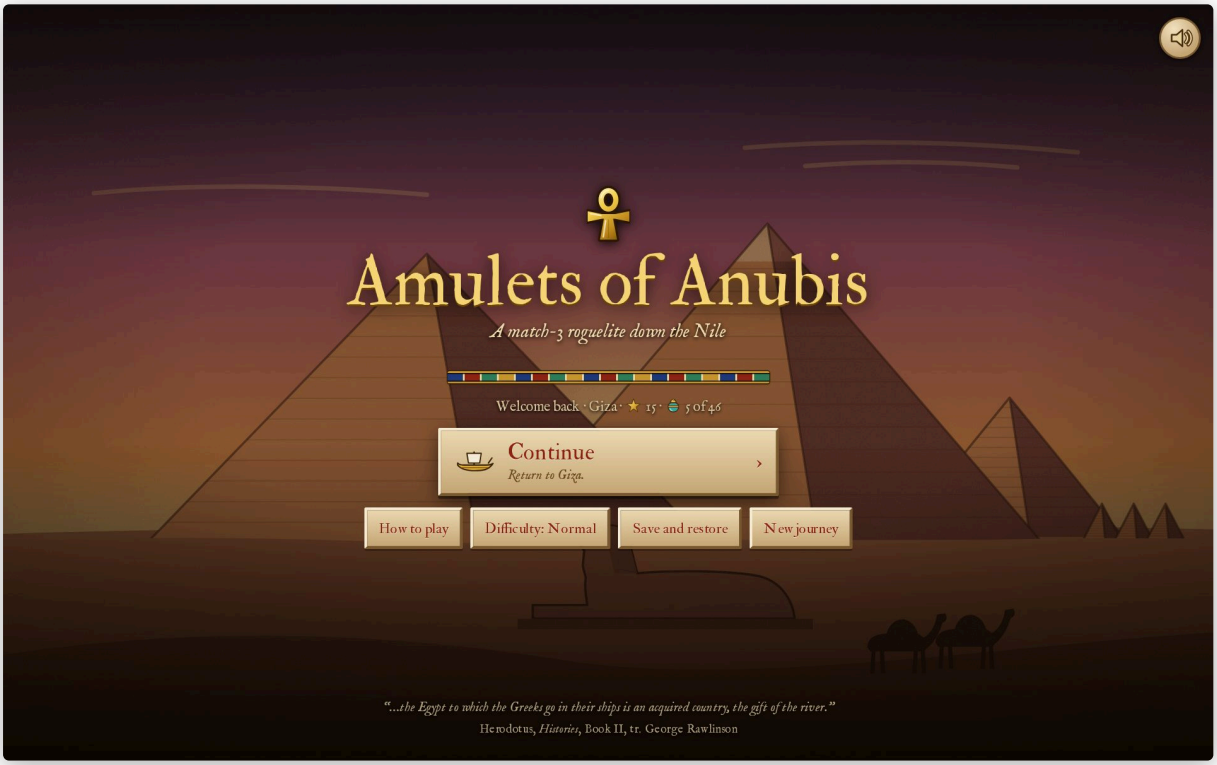
Amulets of Anubis

The manual



Part 1: Making things, for everyone
Part 2: The engine, for programmers

Faience Games · version 0.9.6 · amuletsofanubis.com

Contents

1. Part 1: Making things
    1. Getting ready
    2. The quick way
    3. How the game is put together
    4. The words the game uses
    5. Pictures
    6. Words and numbers
    7. Adding something new
    8. Is it fair? Checking the balance
    9. When the list isn't enough: a little code
    10. If something goes wrong
    11. Sharing your version
2. Part 2: The engine
    1. The shape of it
    2. The build
    3. The source
    4. The rules
    5. The running game
    6. Screens and the interface
    7. The systems
    8. Sound and music
    9. Extending the engine
    10. Saving
    11. Balance and the simulators
    12. Performance
    13. The stylesheet
    14. Testing
    15. Packaging
    16. House style
    17. The Godot version
3. Appendix A: Content reference
4. Appendix B: Building the desktop and Android apps

Part 1: Making things

Part 1 of the manual of Amulets of Anubis. If you're a developer and want to know the inner workings, view part 2.

This part is for anyone who'd like to make the game their own: new stops, new pictures, different words, a harder journey down the Nile or a kinder one. You don't need to be a programmer to be able to do any of that. The vast majority of it is simple text. Some things do need a tiny bit of easy-to-understand code, which I'll walk you through line by line in chapter 9.

Don't worry about breaking anything! Every time you build the game, it checks all of your work first. If something is wrong, it tells you which file and line the mistake is in and what to do about it. Your last working game stays exactly as it was until you've fixed it.

Where else to look

- Part 2 is meant for developers. It explains how the game works on the inside.
- The content reference lists every setting of every kind of file. It's handy when you need the exact name of something.

1. Getting ready

What you'll do: install two programs and build the game. **Time:** about 15 minutes.

You only need two programs for this and both of them are free.

A text editor. This is a program for changing plain text files.

- I would recommend VS Code. It works on every computer and colours the text for you, which makes it much easier to spot a missing quote mark.
- Notepad++ on Windows or TextEdit on a Mac work too. In TextEdit, you have to choose *Format → Make Plain Text* before you save, though.
- Please don't use Word or Pages! They add hidden formatting to the text that the build can't read.

Python 3. This is the programming language the build is written in, so your computer needs it to build the game.

- **On Windows,** double-click `build.bat`. If a window tells you that Python can't be found, download it from python.org. During the installation, make sure to tick the box that says **Add Python to PATH**.
- **On a Mac or on Linux,** it is often already installed. Open the Terminal app, type `python3 --version` and press Enter. If you see a version number such as `Python 3.12.4`, you're good to go.

Building the game

"Building" turns all of the small files into the game.

- **On Windows:** double-click `build.bat`.

- **On a Mac or on Linux:** open Terminal, type `cd` (with a space after it), drag the game's folder onto the window and press Enter. Then type `python3 build.py` and press Enter.

You should see: a few lines of text ending with `Done`. The game has successfully been built! It can be found in `dist/amulets-of-anubis.html`. Double-click it to play.

That one file is the *whole* game. You can copy it anywhere, send it to a friend or keep it for twenty years and it will still work.

> **Keep a copy before you start.** Duplicate the whole folder. If you ever get lost, you can always go back to the copy.

---

2. The quick way

I would recommend the following workflow for quickly and easily adding new things to the game.

1. Let `new.py` start new things for you. Double-click `scripts/new.bat` on Windows, or type `python3 scripts/new.py` on a Mac or Linux. It asks you what you'd like to make and what its id should be. Then it writes a working file in the right folder, with a comment on every line explaining what it does and copies any pictures it needs for you to draw over. The new thing works right away, so you can build and see it in the game before you've changed anything at all!

```
$ python3 scripts/new.py
What would you like to make?

   1. stop          a new stop on the journey, with its own floor, scenery and music
   2. relic         a relic for the museum, found by doing something
   3. event-puzzle  a river event with a small board to play
   4. chamber       a tomb, temple or oasis beside a stop
   5. event-choice  a river event where the player makes a choice
   6. trial         a trial a priest can set, with a boon as the prize
   7. boon          a boon: a held power the player spends
   8. badge         a badge an amulet can fall wearing, with a power
   9. curse         a curse: what a failed trial costs at the next stop
  10. omen          an omen: a hardship braved at a gilded stop, for more reward
  11. stall         something to buy at Anubis's stall
  12. upgrade       a lasting upgrade sold in the Treasury
  13. floor-shape   another floor plan that stops can use
  14. amulet        a brand-new amulet, with a picture to draw over
  15. amulet-set    a new look for the amulets (a recolouring)
  16. floor-set     a new look for the stone floor
  17. frame         a new board frame colour
  18. sparkle       new colours for the sparks that fly off a match

Type a number: 1
Give the stop an id: lowercase letters, numbers and hyphens, like golden-barque: siwa
Copied a starting picture to images/backdrops/siwa.svg
Copied a starting picture to images/boards/siwa.svg

Made content/stops/13-siwa.jsonc: a new stop on the journey, with its own floor, scenery and music.
It works as it is. Next:
  1. Open it in a text editor and change it; the comments say what each line does.
     Draw over the pictures copied into images/ (see images/README.md).
  2. Build (build.bat, or python3 build.py), or leave scripts/watch running.
  3. Double-click dist/try-it.html to try it, with everything unlocked and your real save untouched.
```

`scripts/new.py` *asks what you'd like to make and writes a file that works straight away.*

You can also skip the questions and say it all at once, such as `python3 scripts/new.py relic golden-ankh`. These are the things it can make:

Name	What it makes
`stop`	a new stop on the journey, with its own floor, scenery and music
`chamber`	a tomb, temple or oasis beside a stop
`event-puzzle`, `event-choice`	a river event: a small board to play, or a choice
`trial`	a task a priest can set, with a boon as the prize
`boon`, `badge`, `curse`, `omen`	the powers and hardships of the game
`relic`	a relic for the Museum, found by doing something
`stall`, `upgrade`	something for Anubis's stall, or a lasting upgrade in the Treasury
`amulet`	a brand-new amulet, with a picture to draw over
`floor-shape`	another floor plan that stops can use
`amulet-set`, `floor-set`, `frame`, `sparkle`	new looks, earned by playing

2. Let the game build itself. Double-click `scripts/watch.bat`, or type `python3 build.py --watch` and leave the window open. From now on, every time you save a file, the game is built again all by itself. If you make a mistake, you'll know right away. To stop it, close the window or press **Ctrl+C**.

3. Try it out in try-out mode. Double-click `dist/try-it.html`. This is a special copy of the game meant for testing only (with its own save, separate from your actual game). It unlocks everything and starts at the place that you last edited! It also comes with lots of money and every boon in your inventory, so you can test things more easily. After each change, simply refresh the page.



Try-out mode opens at the stop you just made. The red badge in the corner reminds you that this isn't your real save.

3. How the game is put together

Before you start changing things, it helps to know what each folder is for and what the build does with it.

Folder	What's in it
`content/`	Everything a player sees or reads that isn't a rule: the stops and their history notes, the relics, the river events, the shops, the looks and every word on every screen. There is one small file for each thing. **This is where you will be doing almost all of your work.**
`images/`	Every picture: amulets, scenery, floors, badges and icons. `images/README.md` lists every folder and how its files are named.
`engine/`	The code: the rules of the game, the drawing, the sound and the screens. You only need to open it in chapter 9.

What the build does

When you build, the build (`engine/build.py`) goes through five steps:

Step	What happens
1. Read	It reads every file in `content/` and `images/`.
2. Check	It checks every setting and every floor plan. For instance: can every stone on the floor actually be matched?
3. Connect	It checks that everything a file mentions actually exists. If a stop uses the scarab, for example, there has to be an amulet file for the scarab.
4. Put together	It puts the checked content and the pictures into the code.
5. Write	It writes one file, `dist/amulets-of-anubis.html`, as well as the try-out copy, `dist/try-it.html`.

If step 2 or 3 finds a problem, step 5 never happens. Instead, the build tells you the file, the line and what to do about it.

This is also why the code never needs to know that your new stop exists. Once you've added a stop's file and its pictures, its scenery, its place on the map, its music and its page in How to play all follow by themselves.

New players get to know the game a bit at a time. On a first journey, the game introduces its parts one by one: seals, badges, trials, the stall, river events, cursed badges, tombs and oases, then omens. Each of these arrives with a banner. This means that a new boon or omen of yours only appears in a first journey once its part of the game has arrived. Try-out mode, on the other hand, always has everything turned on.

> **In the engine:** Part 2, chapter 2, *The build* describes each step in detail.

4. The words the game uses

Before we go any further, it helps to know the names the game uses for its pieces, since the files and this manual both use them all the time.

Word	What it is
Stop	One place on the journey down the Nile, like Memphis or Karnak, with its own floor, amulets, scenery and music.
Amulet	The pieces the player swaps. Three or more of a kind in a row make a **match**.
Stone, gild	The squares under the amulets. A match on a stone *gilds* it, which means it turns it to gold. Gild the whole floor to win the stop. *Bare* stone needs one match; *thick* stone needs two.
Special	An amulet made by a bigger match. Four in a row makes a **banded** one, an L or T shape a **ringed** one and five in a row the **winged sun**. Each of them clears more when it's matched.
Badge	A mark that some amulets wear when they fall in. When the amulet is matched, the badge's power is used. Badges with a red ring are **cursed**, meaning their power hurts the player instead.
Boon	A power the player keeps and can use whenever they like, such as the Flood of the Nile.
Trial	A task a priest sets at the start of a stop. Finish it and you win a boon; fail it and the next stop is **cursed**, which makes it harder.
River event	Something that happens between two stops: either a short puzzle on a small board or a choice.
Chamber	A tomb, temple or oasis beside a stop, which opens once the stop has been gilded. Tombs have amulets buried in sand; oases have amulets under water.
Cover	The sand or water an amulet can lie under. It can't be moved until a match beside it has brushed the cover off. Each cover is a file in `content/covers/`.
Seal	Three small challenges at every stop, such as "win without using a boon".
Omen	A hardship a player may choose to face at a stop they have already gilded, in exchange for a bigger reward.
Relic	A lasting reward for a deed, kept in the Museum.
Gold, lapis	The two kinds of money. Gold buys one-time help at Anubis's stall; lapis buys lasting upgrades in the Treasury.
Look	Something that only changes how the game looks: amulet sets, floors, frames and sparkles. Players unlock looks by playing.



Giza in play. Beside the board, you can see the stop's history note, the difficulty and the boons held and the relics found. On the board, some stones have already been gilded. The tube on the right fills with gold sand as the floor is gilded.

---

5. Pictures

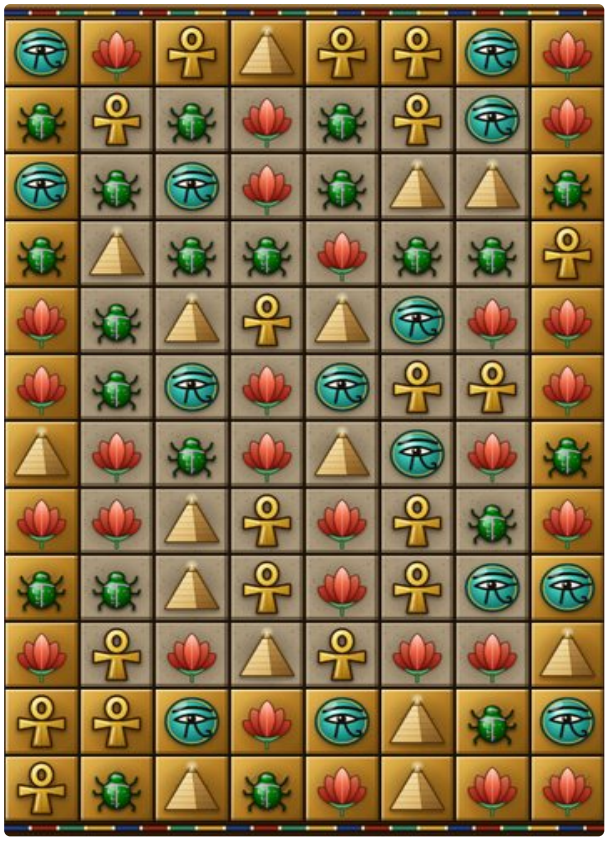
Every picture in the game is a file in the `images/` folder. To change one, simply replace the file with your own (keeping the same name) and build.

Try it first

This is a nice way of seeing how it works:

1. Copy `docs/examples/scarab.png` into `images/amulets/`.
2. Build and play the first stop.

You should see: every scarab is now that picture! A `.png` or `.jpg` is used instead of the `.svg` of the same name and the build lets you know that it did so. Delete the `.png` and build again to get the original back.



Where each picture goes

The pictures that come with the game are **SVG** files. These are drawings made of shapes rather than of dots, so they stay sharp at any size. You can open one in Inkscape (which is free), change it and save it under the same name. You can also drop in a `.png` or `.jpg` with the same name instead.

Folder	What goes in it	Name the file after
`amulets/`	the pieces the player swaps: square, with a see-through background and a small margin	the amulet: `scarab.svg`. A new name is a new amulet, so `golden-falcon.svg` becomes an amulet called `golden-falcon`.
`backdrops/`	the scenery behind a stop and behind the title screen while you're there. It's wide (1600 × 1000) and the edges are cut off on some screens, so keep whatever matters near the middle.	the stop: `giza.svg` is behind Giza
`boards/`	what shows through the gaps of a floor with holes in it: square, 800 × 800	the stop
`floors/<floor set>/`	the squares under the amulets: `bare`, `thick` and `gilded-1` to `gilded-4`	fixed names
`relics/`	a picture that replaces a relic's icon	the relic: `<relic id>.png`
`amulet-sets/<set>/`	an amulet set's own pictures, drawn instead of the usual ones (see New looks)	the amulet: `scarab.png`

Make amulets easy to tell apart. Players recognise amulets by their **colour first and their shape second**. Two amulets of the same colour at one stop are really hard to play, even if their shapes are different. In the same way, keep bare, thick and gilded floor squares easy to tell apart.

Icons

Icons are the small drawings on buttons and screens. They are SVG files, which can be found in `images/icons/`, sorted into these folders:

Folder	What's in it
`dock/`	the buttons along the bottom of the screen
`boons/`	one per boon
`relics/`	one per relic
`menu/`	the tiles of the Menu and the title screen
`codex/`	one per chapter of How to play, named after the chapter
`stall/`	the things Anubis sells that aren't boons (a stall file picks one with `"icon"`)
`ui/`	the ankh (`title-mark`), stars, the seal, the barque (`boat`), Anubis (`keeper`), Apep (`omen`) and the doorways
`map/`	the Nile map (`background`) and its markers

Icons have to stay SVG files. Keep the `viewBox` at the top of the file as it is and give any gradient inside of it a name that no other icon uses (see below).

In another drawing program

The SVGs open in Inkscape, Affinity, Illustrator and Figma as shapes you can change. For other programs, `python3 engine/tools/art-export.py` makes the folder `dist/art-export/`, with every picture as a PDF, the small ones as pixel art at *32* and *64* pixels, as Aseprite files and as sprite sheets for game engines. (The build also makes a large PNG of every picture, in `dist/art-references/`.)

When you save an icon back from a drawing program, check two things (the build checks them too):

1. **It's a clean SVG.** In Affinity or Illustrator, export it as SVG with the text turned into shapes. A picture that isn't written properly stops the build.
2. **Its gradients have names of their own.** A gradient (a colour that fades into another one) has a name inside of the file. Drawing programs call them `_Linear1`, `_Linear2` and so on in every file. Since all icons end up on one page, the second icon would then end up with the first icon's colours. If this happens, the build names both files. Simply rename the gradient in one of them, in both places it appears (`id="manual-making-things--_Linear1"` and `url(#_Linear1)`).

Some pictures are drawn by a script: the scenery (`tools/scenery/`) and many of the amulets (`tools/amulets/draw.py`). If you edit one of those by hand, don't run its script again, or it draws the old picture right back.

A word on picture sizes. I would recommend that you keep pictures small, as each picture makes not only the game bigger but also the amount of data a player has to download if they play in the browser, as well as how much memory (RAM) the game uses. And with how memory prices are going right now, that's not something you want! The build has a built-in warning that lets you know if you add any picture over 2 MB in size.

---

6. Words and numbers

How a content file is written

The vast majority of what you will be adding, such as a new stop, are simple text files in a format called **JSON**. (Technically, it's **JSONC**, "JSON with comments", since plain JSON doesn't allow comments. That's why the files end in `.jsonc`.)

Here is part of a stop's file:

```
{
    // how many moves the player gets
    "moves": 20,
    "name": "Giza",
    "amulets": ["ankh", "scarab", "eye", "lotus"]
}
```

Each line between the curly brackets `{ }` is one **setting**. It is made up of two halves with a colon between them:

- **The left side is the key.** This is the game's internal name for this particular piece of data (`"moves"`). Don't change it, since the game looks for exactly this name.
- **The right side is the value.** This is the *actual* data and the part you will be changing: in this case, how many moves the player gets (`20`).

There are a few different kinds of values:

Kind	Looks like	Used for
Text (a *string*)	`"Giza"`, between double quotes	names, sentences, ids
A number	`20`, without quotes	moves, prices, amounts
Yes or no (a *boolean*)	`true` or `false`, without quotes	turning something on or off
A list	`["ankh", "scarab"]`, between square brackets, with commas in between	several values of the same kind
A group	`{ "stars": 20 }`, between curly brackets	several settings that belong together

There are a few rules to keep the file readable for the build:

- Put a **comma** after each setting, except for the last one.
- Lines that start with `//` are **comments**. These are notes for people and the game simply ignores them.
- Don't delete a bracket, a quote mark or a colon.

- An apostrophe is fine inside of text (*Khufu's*). A double quote (") inside of text is not, since it would end the text early.

If you get one of these wrong, don't worry: the build tells you the file and the line.

Every word on the screens: `content/text.jsonc`

Every word the game's own screens show can be found in one file, `content/text.jsonc`. It is grouped by screen: the title screen, the Menu, the buttons, the shops, the scrolls for winning and losing, How to play and the banners.

```
"win": {
    "title": "{stop} is gilded",
    "lede": "The floor gleamed like the sun at noon.",
```

Change the text on the **right** and leave the key on the left alone.

There are a few special marks you can use:

Mark	What it does	Example
`{stop}`, `{n}` and other words in curly brackets	The game fills in a name or a number here. Keep them, but feel free to move them around within the sentence.	`"{stop} is gilded"` shows "Giza is gilded"
`{n|stone|stones}`	Shows the first word when the number is 1 and the second one otherwise.	"1 stone", "3 stones"
`**bold**`, `*italic*`	Makes words bold or slanted.	
`\n`	Starts a new line.	

If you mistype a key, the build tells you which one it is and suggests the right one.

The names and descriptions of stops, relics, trials, river events, shop items, boons, badges, curses, omens and looks are not in this file, though. Those can be found in each thing's own file.

A stop's name, subtitle and history note

Each stop is one file in `content/stops/`. This is what the beginning of one looks like:

```
{
    "id": "giza",
    "name": "Giza",
    "subtitle": "The house of Khufu",
    "history": "At Giza we stood beneath the Great Pyramid of Khufu ...",
    "moves": 20,
```

Change the text between the quotes. **Once people have played, leave the `id` alone,** since saves remember stops by it.

The history note is written as a travel journal ("we came to Karnak at noon") and its facts should be true: prefer the plainly documented fact to the colourful legend. If a note is too long to fit beside the board, the game shows the first few lines and a "Read on" button, so length is never really a problem. Two or three sentences read best, though.

How many moves a stop gives

In the same file: `"moves": 20`. This is, essentially, the stop's difficulty. More moves is easier. Two or three moves more or less can make quite the difference! The game adjusts this for the difficulty and the size of the screen by itself, so change it a little at a time.

The numbers that tune the game: `content/settings.jsonc`

This file contains the numbers for the whole game, each with a comment beside it. These are the ones you are most likely to change:

Key	What it sets
`stops_open_at_start`	How many stops are open right from the beginning.
`staging`	At which stop each part of the game arrives on a first journey and whether new players get them one at a time.
`stars`	How many moves have to be left over for two stars and for three stars.
`difficulty`	For Relaxed, Normal, Hard and Pharaoh: the moves, how often badges fall and how soon hints appear.
`river_channel`	The colours of the Nile running down an Omega board and its reeds.
`river_event_percent`, `trial_offer_percent`	How often river events and trials come up, out of 100.
`earnings`, `omens`, `seals`, `persistence`	The rewards and how much kinder the game gets after a failed try.

> **Important:** the four groups in `difficulty` also give the difficulties their names. Never rename one, since saves and curses rely on them!

An example: a gentler game for young players. In `settings.jsonc`, open every stop at once and bring in the game's parts all together:

```
"stops_open_at_start": 12,
"staging": {
    "learning_pace": "everything now",
```

Then, in the `difficulty` group, give Normal a few more moves. After you've built it, a new player can go anywhere right from the start. Nothing else changes.

> **In the engine:** the numbers are read by `settings()` in `engine/build.py` and reach the code as `CONTENT.settings`. Part 2, chapter 11 explains which numbers tune what.

Prices

- **Anubis's stall** (help for one stop): `content/stall/`, the `"price"`.
- **The Treasury** (lasting upgrades): `content/treasury/`, the `"prices"` list, with one price per level, like `[300, 600, 1200]`.

7. Adding something new

No matter what you're adding, whether it's a stop, an amulet or a new relic, it's always done the same way:

1. **Make** it with `scripts/new.py` (see chapter 2), or **copy** the nearest existing file by hand.
2. **Rename** the copy with the next number and a new name, if you copied it by hand. `new.py` does this for you.
3. **Change** its `id` and whatever else you need.
4. **Build** and try it out in `dist/try-it.html`.

The number at the front of a file name decides its place in the order. `13-koptos.jsonc` comes after `12-alexandria.jsonc`, meaning Koptos would be the thirteenth stop.

A new stop

`python3 scripts/new.py stop koptos` writes `content/stops/13-koptos.jsonc` and copies Saqqara's pictures for it, so it works right away. Then change these:

Key	What it is
`name`, `subtitle`, `history`, `moves`	See chapter 6.
`amulets`	Four to six amulet ids, from `content/amulets/` or pictures you've added. Make sure they differ in colour **and** in shape.
`floor_plan`	The shape of the floor (see below).
`map_position`	Where the stop sits on the Nile map: `y` goes from about 45 at the coast to 506 at Abu Simbel. `label` decides whether its name is shown on the `"left"` or the `"right"` of the marker.
`music`	A key, a scale and an instrument (`harp`, `lyre`, `oud`, `flute` or `bell`). The game makes up the music as you play, in that key!
`seals`	Three small challenges, like `{"text": "Win without spending a boon", "when": {"win_without_boons": true}}`. `text` is what the player reads and `when` is what the game checks; the content reference lists every condition.

Draw over `images/backdrops/koptos.svg` and `images/boards/koptos.svg` whenever you like, or share another stop's pictures with `"scenery": "giza"`.

The floor plan

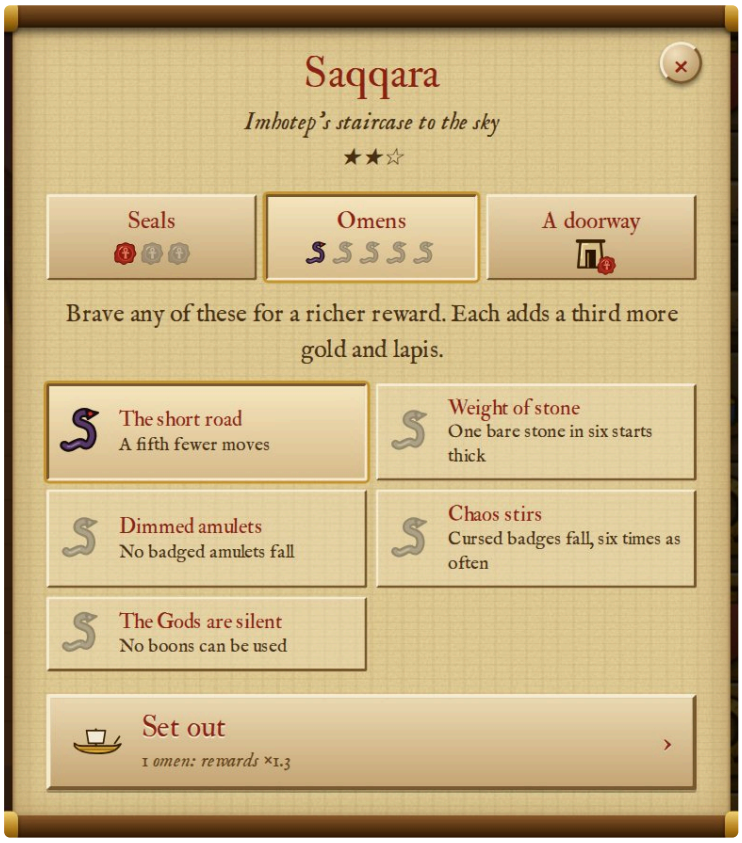
The floor plan is a list of eight lines with eight characters each. Every character stands for one square:

Character	Square
`.`	a gap: there is no square here
`0`	stone that is already gold
`1`	bare stone, which one match gilds
`2`	thick stone, which needs two matches

For instance, a floor in the shape of a ring would look like this:

```
"floor_plan": [
    "..1111..",
    ".122221.",
    "11....11",
    "12....21",
    "12....21",
    "11....11",
    ".122221.",
    "..1111.."
],
```

Every stone has to be matchable. Since a match is three in a row, every stone needs at least two more squares next to it in a line, either across or down. In the ring above, every stone has that. If one of them didn't, the build would stop and tell you so, for example: "the square at row 3, column 1 can never be matched".



A stop's card on the map, once the stop has been gilded. Its three seals come from the stop's file; the omens and the doorway come from files of their own.

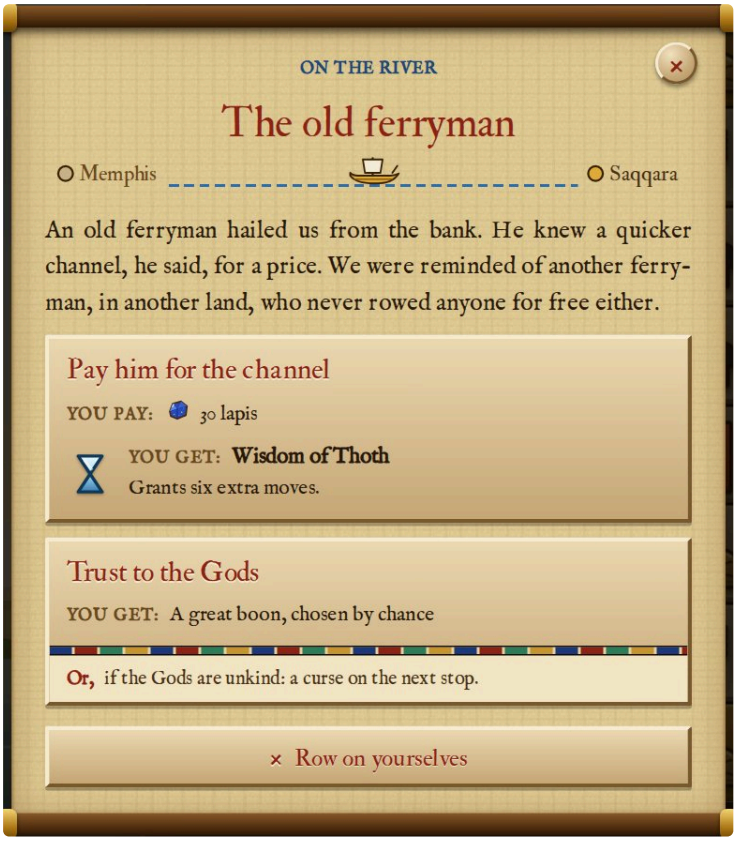
A new amulet

`python3 scripts/new.py amulet golden-falcon` writes `content/amulets/NN-golden-falcon.jsonc` (its name, its plural and what the symbol meant, all shown in How to play) and a picture to draw over in `images/amulets/`. Then add `golden-falcon` to a stop's `amulets`. Until you do that, it won't show up anywhere.

A new river event

River events happen between two stops. There are two kinds of them:

- A **puzzle** (`new.py event-puzzle`) is a small board with a goal: gild the floor, collect a certain number of one amulet or reach a score.
- A **choice** (`new.py event-choice`) offers two or three things to do, each with a cost or a gift. It's a good idea to always include one that simply lets the player sail on.



A new tomb, temple or oasis

`python3 scripts/new.py chamber my-tomb`, then set:

Key	What it is
`at`	The id of the stop it's beside. A stop can only have one chamber.
`name`, `text`	Its name and what the player reads before going in.
`moves`, `floor_plan`	The same as for a stop. In the floor plan, `s` is a stone with an amulet buried in sand.
`reward`	What the player gets for gilding it.
`setting`	`"tomb"` for a dim room lit by torches, `"oasis"` for daylight. In an oasis, use `w` for an amulet under water.

Sand, water and other covers

A **cover** is something that sits atop an amulet and keeps it in place, like sand or water. The amulet can't be moved until the player has broken the cover by making matches. The game has two covers, sand and water, each a file in `content/covers/` with its picture in `images/covers/`.

A cover's file says what it's called, what How to play says about it, the line shown when a player tries to move a covered amulet, the line on a chamber's card, its two letters in floor plans (`s` and `S` for sand), its sound and the colours it bursts into. You can make a new cover (ash, for instance, or amber) simply by making a new file, with no code at all.

A cover can also behave differently:

Key	What it does
`layers`	How many hits it takes to break. With `2`, it needs two matches and a second picture (`<id>-1.svg`) shows it cracked.
`broken_by`	`"beside"`: a match next to it breaks it, like sand. `"on"`: only something that clears its own square does.
`matches`	Whether the amulet under it can still be part of a match (see below).
`spreads`	`true`: after every move that didn't break any of it, it grows over one more amulet next to it, like ivy.

What `matches` means. Imagine a row of three blue amulets where the one in the middle is covered. With `"matches": false` (sand and water), the covered amulet is **stuck**: it doesn't count, so nothing happens. With `"matches": true` (like a net), the covered amulet **still counts**: the row is a match, the two free amulets are cleared and the covered one stays where it is but loses a layer of its cover instead. In short, `false` means "break me before you can use me", whereas `true` means "you can use me and using me breaks me".

The content reference lists every setting a cover has. Tessera's example game, the engine this game is built on, shows all of them in action: amber, a net and ivy.



A tomb's doorway. The picture, the note and the task all come from the chamber's file.

A new relic

`new.py relic`, then set its `name`, its `description` and `found_when`: what the player has to do, like `{"stops_won": 10}` or `{"best_cascade": 5}`. Every condition you list has to be met. Its picture is `images/icons/relics/<id>.svg`, or another icon named in `"icon"`.

An example: a relic for the brave, found by winning twenty stops on Hard (or Pharaoh) and holding 500 lapis at once. The conditions count across all of the player's journeys:

```
"id": "sphinx-seal",
"name": "Seal of the sphinx",
"description": "Win twenty stops on Hard and hold 500 lapis at once.",
"found_when": {"hard_wins": 20, "lapis_held": 500},
"reward": {"lapis": 25}
```

The content reference lists every condition.

A new trial

`new.py trial`, then set its `goal` (clear so many amulets, make specials, make a long cascade, win with moves to spare and so on), its `target` (how many), the words and the boon it gives.

A new boon, badge, curse or omen

These four all work in a similar way. Each of them is a file and each of them picks what it **does** from a list the game already knows. The list is written in the comments of every such file.

- **A boon** (`content/boons/`) is a power the player keeps and uses whenever they like. It has a `name`, a `short` name for its button (about eight letters), a `description` and an `effect` such as `"extra_moves"` with an `amount`. Give it out as a trial's prize, in Anubis's stall or as a river event's reward. Its picture is `images/icons/boons/<id>.svg`.
- **A badge** (`content/badges/`) is a mark an amulet can wear when it falls in. It has a `name`, a few words on how it `looks` (so players can tell badges apart), an `effect` such as `"gild_stones"`, a `weight` (how often it turns up compared to the others) and a glow `colour`. Draw it in `images/badges/`.
- **A curse** (`content/curses/`) is what a failed trial costs at the next stop. It has an `effect` such as `"thick_stones"` and its `strength` on each difficulty. Keep Relaxed at `0`. One that buries amulets can name its `"cover"`, either `"sand"` or `"water"`.
- **An omen** (`content/omens/`) is a hardship a player may choose to face for a bigger reward. It picks from the same list as the curses, with one `amount`.

In the words, write `{n}` wherever the number should go: `"{n} fewer moves"`. If the effect you'd like isn't in the list, chapter *9* shows you how to add one.

Every boon, badge, curse and omen appears in How to play all by itself, with its picture and its words:



Something for the shops

- **Anubis's stall** (`new.py stall`) sells help for a single stop: more moves right now, a reshuffle, a Second wind or a boon.

- **The Treasury** (`new.py upgrade`) sells lasting upgrades, with one price per level and an `effect` such as `extra_moves` or `win_gold`.



New looks

Looks don't change anything but how the game looks. Players earn them by playing. Each look has `unlocked_by`, which describes what the player has to do to unlock it, such as `{"stars": 20}` (earn twenty stars).

- **An amulet set** (`new.py amulet-set`) recolours the amulets with `colour_changes`: `hue_shift`, `saturation`, `brightness` and so on. Shift the colours rather than taking them away, or every amulet ends up looking the same.
- **An amulet set with pictures of its own** draws the amulets completely differently instead, for instance as pixel art. Make a folder `images/amulet-sets/<the set's id>/` and put a picture in it for each amulet, named after the amulet (`scarab.png`, `ankh.png` and so on). Any amulet you leave out keeps its usual picture. For pixel art, draw them small (16 × 16 works nicely) and add `"pixel_art": true` to the set's file, so the game keeps every pixel crisp and square rather than blurring them when it makes them bigger. The game's own four sets of this kind (engraved plates, Naqada pots, Djoser's tiles and cloisonné) are drawn by a script, `tools/amulet-sets/draw.py`, from one simple outline of every amulet in `tools/amulet-sets/shapes.py`. If you add an amulet, give it an outline there too and run the script. Then it gets a picture in all four sets. If you don't, it just keeps its usual picture in them.
- **A look for sale.** Any look can have a price too. Give it `"currency": "lapis"` (or `"gold"`) and `"price": 300`. Customise then shows the price on its card. Players buy it with a tap. Until they close Customise they can change their mind and take it back. If the look has no `unlocked_by`, buying it is the only way to get it. If it has one, players can earn it or buy it, whichever comes first. The engraved plates, Djoser's tiles and cloisonné are sold this way.

- **A floor set** (`new.py floor-set`) is a folder of floor pictures. Check it against every stop's amulets: a blue floor hides a blue scarab!
- **A frame** or **a sparkle** is just a few colours.



Customise shows every look. A locked one says what opens it and how far along the player is.

A new floor shape

`new.py floor-shape` makes a floor plan that several stops can share. Its `move_multiplier` gives harder shapes a few more moves (`1.1` is ten per cent more). To use it, list its `id` in a stop's `shared_floor_shapes`.

Your own sounds and music

Every sound in the game is *made* while you play (the plucked strings, the bells, the click of stone) and so is the music, in each stop's own key and instrument. You don't have to change any of it. But if you have recordings you'd rather use, the game can play those instead.

1. **Sounds:** make a folder called `sounds/` in the game's folder (next to `images/`, not inside it) and put a recording in it for each sound you'd like to replace, named after that sound: `match.mp3`, `gild.wav` and so on. The names are listed at the end of the content reference ("Sounds"). If you misspell one, the build tells you and suggests the right one.
2. **Music:** make a folder called `music/` and put a piece in it named after a stop, such as `giza.mp3`. It then plays, over and over, at Giza. A piece called `default.mp3` plays at every stop that has none of its own. Stops without either keep the music the game makes up.
3. Build and play.

A few things to know:

- `.mp3`, `.wav`, `.ogg` and `.m4a` all work. I would use `.mp3`, since every browser plays it.
- Your recordings go *inside* the game file, like the pictures. So keep them short and small: a whole piece of music can easily make the game several megabytes bigger, which every player then has to download.
- The volume sliders, the mute buttons and the muffling under a scroll work for recordings just as they do for the made-up sounds. What recordings *can't* do is follow the game: the made-up music gets tenser as your moves run out and brighter as the floor turns to gold. Also, the sounds of a match are tuned to the music's chord. A recording always sounds the same.
- Only use recordings you're allowed to share and write down where they came from in `images/CREDITS.md`, just as you would for a picture.

---

8. Is it fair? Checking the balance

What you'll do: let a bot play your stops many times, to find out how hard they are.

Eyeballing the difficulty is bad practice. Your own results will differ from run to run and, as you keep playing, you will naturally get better at the game. Thus, it's best to let a bot play instead: even if it's not particularly smart, it can, at the very least, produce semi-reproducible results. The game comes with one such bot. It plays a stop many times and counts how often it wins. To run it, you need Node.js, a free program for running JavaScript.

1. Install Node.js, if you haven't already.
2. In the terminal, in the game's folder, type:

   ```
   node engine/tools/journey-sim.js 1 60
   ```

   This plays sixty first journeys (`60`) on Normal (`1`, since it starts counting from 0), the way a new player would get to know them.
3. Now wait a minute or two. The bot is quick, but it does play quite a lot of games!

You should see: for each board size, how often the bot won each stop on its first try, in per cent, one number per stop in the order of the journey, followed by the average.

What to aim for: I aimed for a win rate of about **65 to 75 per cent** for the bot on Normal, with the early stops a little easier than the later ones. In real-world terms, this seems to equate to a win rate of about 25 to 75 per cent for players, depending on their skill level. If your stop is far off, change its `moves` by one or two and run the bot again.

For a river event puzzle or a chamber, `node engine/tools/events-sim.js` does the same. Here, the aim is somewhere between half and nine in ten (50 to 90 per cent).

---

9. When the list isn't enough: a little code

All that we've worked on thus far has been things that the engine has built into it: a boon's `effect`, a badge's `effect` and a curse's `effect` are all one name from a list the engine already knows. However, there may be times where you wish to add something that I did *not* think of adding to this game. If that is the situation you're finding yourself in, then you'll want to read this chapter.

You will be required to write a bit of code, namely a few lines of **JavaScript** (the programming language the engine is written in)! But you don't have to be an actual, proper programmer to do any of this. Most new things start out as a copy of something that already does something similar. On top of that, I'll explain every line you will come across.

The bits of JavaScript you'll come across

Before we look at any real code, here are the handful of things you will see in it again and again. You don't have to learn them by heart; just come back to this section whenever a line doesn't make sense to you.

`core`: the game that is being played. Almost every rule is handed something called `core`. This is one big bundle containing everything about the stop that is being played right now: the floor, the amulets on it, how many moves are left and so forth. A rule is given `core` so that it can look at the board and change it. (The name comes from the fact that it is the *core* of the game: the part that knows the rules, without any of the drawing or the sound.)

The dot: "the part of … called …". A dot opens up a bundle and picks one thing out of it:

```
core.movesLeft   // the part of core called movesLeft: the number of moves left
core.floor       // the part of core called floor: the stone of every square
```

Square brackets: "item number … of a list". Many things in `core` are lists with one entry per square of the board. Square brackets pick one entry out of such a list:

```
core.floor[12]   // the stone on square number 12
core.floor[sq]   // the stone on square number sq, whichever square sq is
```

Names for things. `const` gives a name to something you have worked out, so that you can use it again further down. `let` does the same for something that is going to change later on:

```
const col = colOf(sq);   // col is now the column of square sq
let n = 0;               // n starts at 0 and can be changed later
```

Changing a number. A single = means "becomes". There are also a few shorthands:

Written	Means
`core.floor[sq] = 1;`	the stone on square `sq` becomes `1` (bare)
`core.movesLeft += 3;`	three more moves (the moves left become what they were, plus `3`)
`core.floor[sq]--;`	one less: the stone loses a layer

Asking questions. These give you `true` or `false`, which you'll need for choosing squares and for `if`:

Written	Asks	For example
`===`	is it exactly the same? (three = signs, since one already means "becomes")	`core.floor[sq] === 1`: is this a bare stone?
`!==`	is it different?	`core.floor[sq] !== 0`: is it *not* gilded yet?
`>`, `<`	is it more? less?	`core.floor[sq] > 1`: is it thick?

Written	Asks	For example
`&&`	and: are both true?	`core.mask[sq] && core.floor[sq] > 1`: is it part of the floor *and* thick?
`||`	or: is at least one of them true?	
`!`	not: turns true into false and the other way round	`!thick.length`: is the list empty?

Doing something: a function. A function is a set of steps that you can hand things to. The arrow `=>` separates what the function is given from what it does, so you can read it as "given ..., do ...":

```
(core, n) => pickSome(bareStones(core), n)   // given core and n, pick n bare stones
sq => thicken(core, sq)                      // given a square, make it thick
```

The names before the arrow are what the function is given; they're simply names that the function uses for them while it runs. When there is only one of them, the brackets can be left out, as in `sq => ...`. When a function needs more than one step, its steps go between { and }, one per line:

```
async (b, sq) => {
    const col = colOf(sq);
    ...
}
```

Groups of named parts, between { and }, just like in a content file. The big lists in this chapter (`BOON_EFFECTS`, `BADGE_POWERS` and `HARDSHIPS`) are groups like this, with one entry per effect. An entry can itself contain functions:

```
thick_edges: {                // the entry's name, which a content file uses
    board: (core, n) => ...,  // its part called board: a function
},
```

Going through a list. A list comes with a few ready-made helpers of its own, which you call with a dot. Each of them is handed a small function that it runs once for every item in the list:

Written	Does	For example
`list.filter(test)`	keeps only the items for which the test is `true`	`squares.filter(sq => core.floor[sq] === 1)`: only the bare stones
`list.forEach(step)`	does the step once for every item	`squares.forEach(sq => thicken(core, sq))`: make every one of them thick
`list.length`	how many items there are	`thick.length`

Repeating and choosing.

```
for (let row = 0; row < ROWS; row++) { ... }   // runs the lines inside once for every row
if (core.movesLeft < 5) { ... }                // runs the lines inside only if fewer than 5
moves are left
```

Waiting. `await settle()` waits until the amulets have stopped moving before carrying on. A function that has to wait like this is marked with `async` at its beginning.

The small print. A line usually ends with a `;`, much like a sentence ends with a full stop. Everything after `//` is a comment: a note for people that the game ignores.

The board, as the code sees it

To the code, the board is essentially **one long list of squares**, row after row. A board that is eight squares wide numbers them like this:

```
 0  1  2  3  4  5  6  7
 8  9 10 11 12 13 14 15
16 17 18 ...
```

A square's number is called `sq`. For each square, there are three lists that describe what is there:

The list	What it tells you	For example
`core.mask[sq]`	whether this square is part of the floor at all	`false` for a gap
`core.floor[sq]`	how many layers of stone are left	`0` gilded, `1` bare, `2` thick
`core.cells[sq]`	the amulet on it	its kind and any special, badge or cover; or nothing

`COLS` is the number of columns and `ROWS` the number of rows. Boards are often taller than they are wide, so never assume the two are the same.

That said, you will rarely have to work out rows and columns yourself, since the **helpers** below do that for you.

Helpers: the words the rules are written in

A **helper** is a small, ready-made function with a plain name that does one thing every rule needs: finding squares, choosing some of them or changing them. A new rule is usually just two or three helpers in a row, so it reads almost like it would in plain words.

The helpers can be found at the top of `engine/src/01-core.js`, under "helpers". You can use them anywhere in the game's code.

Finding squares

Helper	What it gives you
`rowOf(sq)`, `colOf(sq)`	the row or the column of a square
`squareAt(row, col)`	the square at that row and column, or `-1` if that is off the board
`neighbours(sq)`	the four squares next to it: above, below, left and right
`around(sq, reach)`	every square near it, including itself. A reach of 1 gives the 3 × 3 around it; 2 gives 5 × 5.
`rowSquares(row)`, `colSquares(col)`	every square in a row or in a column
`diagonalSquares(sq)`	every square on both diagonals through it

Finding what's on them

Helper	What it gives you
`bareStones(core)`, `thickStones(core)`	the bare stones or the thick ones
`stonesLeft(core)`	every stone that hasn't been gilded yet
`gildedSquares(core)`	every gilded square
`edgeOfFloor(core)`	the squares around the edge of the floor
`plainAmulets(core)`	the amulets without a special, badge or cover
`coveredAmulets(core)`	the amulets under any cover
`coveredAmulets(core, 'water')`	the amulets under one particular cover
`amuletsOfType(core, type)`	every amulet of one kind
`squaresWhere(core, (sq, tile) => ...)`	anything else. You write the test yourself, as a small function: it is given each floor square and the amulet on it. It then answers `true` or `false`. For example, `squaresWhere(core, (sq, tile) => tile && tile.special)` gives every special amulet.

Choosing and changing

Helper	What it does
`pickSome(list, n)`	picks `n` of the list at random
`pickOne(list)`	picks one of the list at random
`thicken(core, sq)`	makes a bare stone thick
`coverAmulet(core, sq, 'sand')`	puts an amulet under a cover
`uncover(tile)`	frees an amulet from its cover
`coverAmulets(core, n, 'sand')`	covers `n` plain amulets at random, but never so many that the player has no move left
`makeSpecial(core, sq, 'bomb')`	turns an amulet into a special (`h`, `v`, `bomb`, `star` or `sun`) or gives it a badge

An example. This is a whole curse written with helpers. It is the engine's `thick_edges`, which makes "some bare stones along the edge of the floor start out thick". When a stop with this curse begins, the engine hands its `board` function two things. The first is `core`, the board that has just been filled. The second is `n`, the curse's strength (how many stones to thicken). Written out one step per line, it looks like this:

```
thick_edges: {
    board: (core, n) => {
        const edge = edgeOfFloor(core);
        const bare = edge.filter(sq => core.floor[sq] === 1);
        const chosen = pickSome(bare, n);
        chosen.forEach(sq => thicken(core, sq));
    },
},
```

Line	What it does
`thick_edges: {`	The curse's name. A curse file picks it with `"effect": "thick_edges"`.
`board: (core, n) => {`	The part that changes the board. Given `core` and `n`, it does the steps below.
`const edge = edgeOfFloor(core);`	Takes every square along the edge of the floor and calls that list `edge`.
`const bare = edge.filter(sq => core.floor[sq] === 1);`	Goes through `edge` and keeps only the squares whose stone is exactly 1, the bare ones. That list is called `bare`.
`const chosen = pickSome(bare, n);`	Picks `n` of those at random and calls them `chosen`.
`chosen.forEach(sq => thicken(core, sq));`	For every chosen square, makes its stone thick.
`},`	The end of the function, then the end of the entry.

In the engine itself, the same curse is written in one go, without the names in between. It does exactly the same thing; it's just shorter. You read it from the inside out:

```
thick_edges: {
    board: (core, n) =>
        pickSome(
            edgeOfFloor(core).filter(sq => core.floor[sq] === 1),
            n,
        ).forEach(sq => thicken(core, sq)),
},
```

Both ways of writing it work, so use whichever you find easier to read!

No curse of this game uses `thick_edges` yet, but a curse file with `"effect": "thick_edges"` would.

Reading a boon that already exists

Before making something new, it helps to read something that already works. A boon file names *which* effect it uses, with `"effect"`. The effect itself is an entry in the list `BOON_EFFECTS`, in `engine/src/game/07-trials-boons.js`. This is the one behind the Wisdom of Thoth:

```
// n more moves
extra_moves: {
    target: false,
    amount: 6,
    use: async b => {
        core.movesLeft += b.n;
        updateHUD();
        sfx('moves');
        boonPopup(b, false, { x: COLS / 2, y: ROWS / 2, life: 1.6, size: 0.6, col: '#bfe8ff' });
        rings.push({ x: COLS / 2 - 0.5, y: ROWS / 2 - 0.5, life: 1, big: true });
    },
},
```

Line	What it means
`// n more moves`	A comment: a note for people, saying what the effect does.
`extra_moves: {`	The effect's name, which a boon file uses in `"effect"`.
`target: false,`	The player doesn't choose a square. With `true`, the board glows gold and waits for the player to tap a square (as in the picture below), which is then handed to `use` as `sq`.
`amount: 6,`	The number used when a boon file doesn't give an `amount` of its own.
`use: async b => {`	What happens when the player uses the boon. The function is given `b`, the boon itself; `b.n` is its amount. It's marked `async` because boons are allowed to wait (this one doesn't need to).
`core.movesLeft += b.n;`	**The rule itself:** the moves left go up by the boon's amount.
`updateHUD();`	**From here on, it's the show.** This updates the numbers at the top of the screen, so the player sees the new moves.
`sfx('moves');`	Plays the sound called `moves`.
`boonPopup(b, false, { ... });`	Floats the boon's words (its `popup`) up over the board. `false` means "there was something to do", so the normal words are used. The group in `{ }` says where and how: `x` and `y` are the position, counted in squares from the top left (`COLS / 2` is halfway across the board), `life` how many seconds the words stay, `size` how big they are and `col` their colour.
`rings.push({ ... });`	Adds a ring of light to the list of rings the game draws, in the middle of the board. `big: true` makes it a big one.
`},`	The end of `use`, then the end of the entry.

Every effect is built like this: first comes the rule, then the show.



A boon with `target: true`*, waiting for the player to choose a square.*

Making your own: the Blessing of Hapi

What you'll do: make a new boon, the Blessing of Hapi, which turns every thick stone in a column into bare stone.
Time: about 20 minutes.

Hapi was the God of the Nile's yearly flood, the water that softened the fields. So let's make a boon in His name that **softens every thick stone in the column the player chooses**, turning it into plain bare stone. None of the effects in the list does this yet. The Cord of Seshat, however, is pretty close: it works on the **row** the player chooses. So we copy the idea and change it.

Step 1: adding the effect

1. Open `engine/src/game/07-trials-boons.js`.
2. Find `cord_row:` in `BOON_EFFECTS`. This is the Cord of Seshat's effect.
3. Right above it, add these lines:

```
// every thick stone in the chosen column becomes bare stone
thin_column: {
    target: true,
    use: async (b, sq) => {
        const col = colOf(sq);
        const thick = colSquares(col).filter(j => core.mask[j] && core.floor[j] > 1);
        thick.forEach(j => {
            core.floor[j] = 1;
            flashes.set(j, 1);
        });
        bgDirty = true;
        beams.push({ dir: 'v', idx: col, life: 1 });
        sfx('blessing');
        boonPopup(b, !thick.length, { x: col + 0.5, y: 0.8, life: 1.6, size: 0.46, col: '#bfe8ff'
});
        updateHUD();
        await settle();
    },
},
```

This is what each line does:

Line	What it does
`thin_column: {`	The name a boon file will use in `"effect"`.
`target: true,`	The player chooses a square first.
`use: async (b, sq) => {`	What happens when it's used. Because of `target: true`, the function is given the square the player chose, as `sq`, as well as the boon, `b`. It's `async` because it waits at the end.
`const col = colOf(sq);`	Finds the column of the chosen square and calls it `col`.
`const thick = colSquares(col).filter(j => ...);`	Takes every square of that column from top to bottom. It then keeps only those that are part of the floor **and** thick (`core.mask[j] && core.floor[j] > 1`). That list is called `thick`. The squares are called `j` here, since the name `sq` is already taken by the chosen square.
`thick.forEach(j => { ... });`	Does the two lines inside once for each of those squares:
`core.floor[j] = 1;`	...the stone becomes bare (**the rule**)...
`flashes.set(j, 1);`	...and a flash of light appears on it.
`bgDirty = true;`	The floor is painted only once and then kept, since that's much faster. This line tells the game that the floor has changed, so it has to be painted again.
`beams.push({ dir: 'v', idx: col, life: 1 });`	Draws a beam of light down (`'v'`, for vertical) column number `col`.
`sfx('blessing');`	Plays the sound called `blessing`.
`boonPopup(b, !thick.length, { ... });`	Shows the boon's words over the column. `!thick.length` is `true` when the list `thick` is empty, in which case the boon's "nothing to do" words (`popup_when_nothing`) are shown instead.
`updateHUD();`	Updates the numbers at the top of the screen.

Line	What it does
`await settle();`	Waits until the board is still before the player can move again.

There's no `amount` line, since this boon has no number to set. Notice also that the rule itself is just one line (`core.floor[j] = 1;`); the rest makes it *feel* like something happened. A boon without a show works, but it feels broken.

Step 2: making the boon

1. Type `python3 scripts/new.py boon hapi`. It writes `content/boons/11-hapi.jsonc`, a working boon for you to change.
2. Set:

```
"name": "Blessing of Hapi",
"short": "Hapi",
"description": "Softens every thick stone in the column you choose.",
"effect": "thin_column",
"popup": "Hapi softens the stone",
"popup_when_nothing": "No thick stone there",
```

3. Delete the `"amount"` line and keep the words free of `{n}`, since this boon has no number. The build reads the effect names from the code, so it accepts `thin_column` as soon as your entry is there.

Step 3: a picture and a way to be found

Draw `images/icons/boons/hapi.svg` (copy another boon's icon to start from), or write `"icon": "flood"` to borrow one for now. Then give the boon out somewhere: as a trial's prize (`"boon": "hapi"` in a trial file), in Anubis's stall or as a river event's reward.

Step 4: checking it

1. Build.
2. Open `dist/try-it.html`. Try-out mode gives you one of every boon, so Hapi is in the row beside the board.
3. Pick a stop with thick stone, click Hapi and click a column.
4. Run the **smoke test**, which plays the game for a moment and tells you if anything went wrong:

```
node engine/tools/screenshots/smoke.mjs
```

You should see: the thick stones in the column turn bare and the smoke test ends with `No errors.` Your new boon works!

Finally, add a line for `thin_column` to the list of effects in the comment at the top of the boon files and in `scripts/new.py`'s boon template, so the next person knows it exists.

Useful things to call

These are the pieces the existing entries use and the ones you are most likely to want.

To...	Write
change the moves	`core.movesLeft += 3;`
take a layer off a stone	`core.floor[j]--;` and then `bgDirty = true;`
clear amulets as if they had been matched (gilding under them, with cascades after)	`await cascade(null, [j1, j2, ...], true);`
light up a square	`flashes.set(j, 1);` or `burst(column, row, 4);` for sparks
draw a beam along a row or column	`beams.push({ dir: 'h', idx: row, life: 1 });` (`'v'` for a column)
send golden orbs flying from one square to another	`orbs.push({ from: j, to: j2, t: 0, dur: 0.5 });`
play a sound	`sfx('blessing')`, `sfx('crack')`, `sfx('bomb')`, `sfx('moves')` …
show the boon's words on the board	`boonPopup(b, nothingHappened, { x, y, life, size, col });`
update the numbers at the top	`updateHUD();`
wait until the board is still	`await settle();`

When it doesn't work

- **The build says the effect is unknown.** Check that the spelling is the same in the boon file and in the code and that your entry is inside of `BOON_EFFECTS` (between its `{` and the closing `};`).
- **The game shows a blank page.** A bracket or a comma is probably missing. Open the browser's console (F12) and read the red message: it names the line.
- **Nothing seems to happen.** Did the floor change but not its picture? Then you need `bgDirty = true;`. Did nothing at all happen? Check your `.filter`: maybe no square passed its test.

The other lists

The same idea works for the other kinds, too. Each list has a comment above every entry saying what it does, so reading two or three of them is the best way to start.

To make a new...	Add an entry to	In	Further reading
boon effect	`BOON_EFFECTS`	`engine/src/game/07-trials-boons.js`	Part 2: a new boon effect
badge power	`BADGE_POWERS` (the rule) and `BADGE_SHOWS` (how it looks)	`engine/src/01-core.js` and `engine/src/game/21-moves.js`	Part 2: a new badge power
curse or omen effect	`HARDSHIPS`	`engine/src/01-core.js`	Part 2: a new hardship
special's power	`SPECIAL_POWERS` and a rule in `clearStep` for what makes it	`engine/src/01-core.js`	Part 2: a new special

To make a new...	Add an entry to	In	Further reading
helper	the "helpers" section, next to the other helpers of its kind	`engine/src/01-core.js`	Part 2: the board
cover	nothing: a file in `content/covers/`		Sand, water and other covers
trial goal	`TRIAL_GOALS` and count it in `trialProgress()`	`engine/build.py` and `engine/src/game/07-trials-boons.js`	Part 2: a new trial goal

Keep the rules and the show apart. Rules belong in `engine/src/01-core.js`, which doesn't have any drawing or sound in it, since the bot from chapter 8 plays that file on its own to test the balance. How a badge looks and sounds goes in `engine/src/game/21-moves.js`.

After changing a rule, type `node engine/tools/rules-test.js`. It tries every helper, cover, badge power and hardship on a small board of its own and tells you if one of them no longer does what it should.

10. If something goes wrong

Things will go wrong now and then; that's perfectly normal! These are the situations you're most likely to run into and what to do about them.

The build stops and says a file has a problem.

1. Read the line it prints. It names the file, the line, the text and what to do about it.
2. Fix that one thing.
3. Build again.

```
$ python3 build.py
Building Amulets of Anubis...
  pictures: 126 from images/, 114 icons

1 problem in the content files. Nothing was built; the last good game file is unchanged.

  x content/stops/01-memphis.jsonc, line 11: this is not valid content (Expecting ',' delimiter).
      "amulets": [
      A comma is probably missing at the end of the line before this one (or a " is missing).

Fix the first one, build again and repeat. The manual, docs/manual/part-1-making-things.md ("If something goes
wrong"), can help.
```

Here, the comma at the end of a stop's `"moves"` *line was missing. Nothing is built until it's fixed and the game you had before keeps on working in the meantime.*

These are the most common slips:

1. a missing comma between two settings, or a missing quote mark,
2. a key the build doesn't know (it suggests the nearest one it does know),
3. a bracket that was deleted by accident when copying a file.

Try-out mode shows something odd. Try-out mode has a save of its own. To start it afresh, open *Menu* → *Save and restore* → *Complete reset* while in try-out mode. This only clears the try-out save, not your real one.

The game shows a blank page, even though the build said `Done`.

1. Press **F12** (on a Mac, **⌥⌘I**) to open the browser's developer tools.
2. Click on the **Console** tab.
3. Read the red message. It names the problem and usually the line as well.

When in doubt, change one thing at a time: change it, build, check it and then move on to the next one.

---

11. Sharing your version

Once your version is ready to be played by other people, there are a few ways of getting it to them.

- **The file itself.** `dist/amulets-of-anubis.html` is the whole game. You can send it to people, put it on a USB stick or put it on any website. It doesn't need a server and doesn't load anything from the internet.
- **As an app.** `./build.sh` also makes the Android app and the desktop apps for Windows, macOS and Linux. The build guide explains what each of them needs.
- **The licence.** The code is licensed under the GNU General Public License, version 3 or later and the pictures are in the public domain (CC0). This means you may share your version and its code stays free for anyone to read and change. The README has the details.

Part 2: The engine

Part 2 of the Tessera manual, for developers: how the engine works on the inside and how to change it. If you'd like to make a game with content files and pictures, part 1 is what you're looking for.

This part is meant for people who'd like to change the engine itself: add a new kind of rule, a new screen or a new tool. I assume you've read chapter 4 of part 1 ("How a game is put together") and that you know a bit of JavaScript and Python. (Part 1 can be found beside this part in Tessera and in a game's own `docs/manual/`.)

All the paths below are written the way they look inside a game, with the engine in its `engine/` folder. In the Tessera repository itself, the same files can be found at the top instead.

1. The shape of it

A game made with Tessera is **one HTML file** that is meant to keep working, unchanged, for decades. That's the most important thing about the engine. Four rules follow from it and every change to the engine has to keep all of them:

- **One file.** The build writes `dist/<file>.html` (the `file` in the game's `edition.jsonc`). That file is the game. It runs from `file://` with no server.
- **No network.** No CDNs, web fonts, analytics or `fetch`. The fonts are embedded. `grep -c "https://" dist/<file>.html` must print `0`.
- **No dependencies.** Plain HTML, CSS and JavaScript, no packages, frameworks or bundlers. The build needs Python 3; the game needs a browser. Pictures are files; sounds and music are synthesised, unless a game brings recordings of its own.
- **Old WebViews.** A game may be wrapped in an Android app, from Android 7 up. Avoid what WebView lacks, or give it a fallback.

The organising idea is that **content and code are separate**. Everything a player sees or reads that isn't a rule is content, one file per thing in the game's `content/`. The rules, the drawing, the sound and the screens are code, in `engine/src/`. `engine/build.py` is the seam between the two: it checks the content and then puts it into the code.

```
build.py                   (the game's) runs engine/build.py for this game
edition.jsonc              the game's name, file, save keys and code prefix
content/                   the content, one file per thing, plus text.jsonc and settings.jsonc
images/                    every picture; icons in images/icons/<group>/
web/                       (optional) the game's own look: css/, fonts.css
engine/build.py            checks the game, joins engine/src/ and engine/web/, writes dist/
engine/src/01-core.js      the rules, with no browser code (the simulators run it)
engine/src/00-open.js      loads and migrates the save
engine/src/02-pictures.js  loads the pictures before the game starts
engine/src/03-themes.js    per-stop themes, built from the stops
engine/src/04-boards.js    frame colours, amulet sets, floor squares
engine/src/game/           the running game, one file per part (listed in its README)
engine/web/shell.html      the page, with holes the build fills
engine/web/css/            the stylesheet, one file per part of the screen, in a plain look
engine/web/fonts.css       the two typefaces, embedded
engine/tools/              simulators, the smoke and edge tests, the formatter
engine/example/            the example game
engine/godot/              the Godot version (chapter 17)
```

The engine and the game. Everything in `engine/` is the engine and makes any game from any folder of content. Nothing in it names a place, a God or an amulet: it reads them from the game's folder, the one that holds `edition.jsonc` (how it finds that folder is in Tessera's README). That goes for the look too: the engine's own stylesheet is plain and a game brings its own colours, textures, typefaces and tab picture (chapter 13). So a change to the engine goes in a commit of its own that touches only `engine/`. A change to a game's content goes in the next one. (Tessera's own `docs/games.md` explains why and how fixes travel between Tessera and its games.)

---

2. The build

`engine/build.py` runs in this order and **writes nothing if any step fails**, so a broken content file never ships a broken game.

1. **Name lists from the code.** Some names a content file may use are defined in the code and the build reads them from it by pattern: boon effects (`BOON_EFFECTS` in `engine/src/game/07-trials-boons.js`), badge powers (`BADGE_POWERS`) and hardships (`HARDSHIPS`, both in `engine/src/01-core.js`). Others are listed at the top of `engine/build.py`: instruments, scales, board modes, trial goals, upgrade effects, condition names, special pictures, colour changes. Add to those by hand when you add one in the code.
2. **The edition.** `edition.jsonc` is checked (its keys, the starting looks it names) and reaches the code as `CONTENT.edition`.
3. **Pictures and icons.** Every file in `images/` goes into `PICTURES`; every SVG in `images/icons/<group>/` into `ICONS`, cleaned of comments.
4. **Content.** Each folder in `content/` is read in file-name order. Files may carry `//` comments and trailing commas, which are stripped with line positions kept, so errors point at the right line. Each kind has a validator (`stop()`, `boon()`, `badge()` and so on) built on a small `Check` class that reports by file, field and line and suggests the nearest name for a typo. The `KINDS` table ties folders to validators.
5. **References.** Anything that names another thing (a stop's amulets and floor shapes, a relic's condition, a stall item's boon, a trial's boon) is checked once everything is read (`LATER`).
6. **Settings and text.** `settings()` parses `content/settings.jsonc` into the numbers the code reads as `CONTENT.settings`. `content/text.jsonc` is checked against the code: every `T('…')` and `{{…}}` must have a key. Keys the code builds from an id (`T('codex.tabs.' + id)`) can't be found by searching, so they are listed

in `TEXT_FAMILIES`.

7. **Joining.** `engine/src/00-open.js`, `02-pictures.js`, `03-themes.js`, `04-boards.js` and then every file in `engine/src/game/` in name order are joined and `engine/web/shell.html` wraps them in one function (`(() => { … })();`), so the game's names stay its own. `engine/web/css/` is joined in name order, with the game's own stylesheet files in their place (chapter 13).
8. **Filling the holes.** The build puts the checked content, pictures, icons, styles, fonts and code into its placeholders: `/*CONTENT*/null`, `/*IMAGES*/null` and `/*ICONS*/null` in the code (a space before the `null` is fine), `/*STYLES*/`, `/*FONTS*/`, `/*CORE*/` and `/*UI*/` in `engine/web/shell.html`. **These placeholders are load-bearing**: if one of the code's three is missing, the build stops and says so.

`--check` validates without writing. `--watch` rebuilds on save. `--content-json` prints the checked content and nothing else, which is how the simulators get it (`engine/tools/load-core.js`). The build also writes `dist/try-it.html`, which opens try-out mode at the file changed last and, when `rsvg-convert` (librsvg) is installed, `dist/art-references/`: every picture in `images/` as a PNG, in the same folders, for showing the art outside the game (`engine/tools/art-references.py`; only changed pictures are drawn again).

The docs that follow the code

Two parts of the engine's docs are written by the build itself, so they can never fall behind:

```
python3 engine/build.py --docs
```

writes the appendix of the content reference ("every name the build knows") from the names in the code. It also writes a map of the code from the note at the top of every file. It then checks the rest of the reference, which is written by hand: every setting of every kind of content file needs a row in its table, "yes" and "no" in the Needed column have to agree with what the build really insists on (it tries each file without each setting to find out) and the tables of boon effects, badge powers, hardships and conditions have to list exactly what the code knows. If anything doesn't match, it says what and where. Its code can be found in `engine/tools/reference.py`.

So when you add a setting, a boon effect or a new file, run it once and add the row it asks for. When you change what a file is for, change the note at its top: that's what the map shows.

The manual as a website and a book

```
python3 engine/tools/docs/build.py --pdf
```

makes the manual into a website in `dist/docs/` (open `index.html`; it needs no server and no internet) with a search over every page, and into a PDF beside it, on A4 and on A5 (smaller pages, which read better on a phone; `--a4` or `--a5` makes only one). Which pages there are, their order and which of them go into the PDF are listed in `docs/site.json`. The looks are in `engine/tools/docs/looks/` and `--look` picks one; a game can bring its own (a stylesheet in its folder, named in `site.json`). The `dist/docs/` folder can be copied anywhere as it is, to a web server for instance. The PDF is printed by `engine/tools/screenshots/pdf.mjs`, so it needs Playwright like the screenshots.

3. The source

Everything in `engine/src/` (apart from `01-core.js`, which is loaded on its own) runs inside **one function**, top to bottom, in file order. Each file is complete JavaScript on its own, so an editor can read it. So:

- **The numbers are the order.** A `let` or `const` used while the game starts (by `fit()`, or anything the boot calls) must be declared in an earlier file than its first use, or the game throws on load (a temporal dead zone). Functions are hoisted and don't mind.
- **A new file** needs only a name with the right number.
- **Each file starts with a note** saying what it is for, its main functions, where they are called from and what it changes in the save. Read the note first; keep it up to date when you change the file. An editor that reads `jsconfig.json` (VS Code, Zed, Gram) can also follow any name to where it is defined.
- **To find something,** search `engine/src/game/` for its name. Headings like `// ---------- sound ----------` mark the parts inside a file.
- **HTML in the code** is written in `` `html… strings, laid out one element per line and indented like the page it makes. The html helper (02-screen.js)` `` drops the line breaks and the indentation after them, so the page gets the same HTML as if it were written on one line. A space that should show goes before the line break.
- **One style.** The code and the stylesheet are kept in one style by Prettier, with the settings in `.prettierrc`: tabs, lines up to 110 characters, one CSS property per line. `engine/tools/format.sh` lists the files out of style and `engine/tools/format.sh --write` fixes them (it fetches Prettier the first time; the game never needs it). An editor with a Prettier plugin reads the same settings, so format on save agrees. `engine/src/game/README.md` lists what each file holds.

`engine/src/01-core.js` contains the rules and has **no browser code** at all: no `document`, `window`, canvas or audio. That is what lets the simulators run the real rules in Node. Keep it that way.

The code is in one plain style throughout: tabs, one statement per line, spaces after commas and around operators, braces on the same line, single quotes, lines up to about 110 characters, a blank line between functions.

4. The rules

The board

`Core` (in `01-core.js`) holds three flat arrays of length `COLS × ROWS`:

- `mask[sq]`: is this square part of the floor;
- `floor[sq]`: layers of stone left (`0` gilded, `1` bare, `2` thick);
- `cells[sq]`: the amulet there, `{type, special, id}`, with `cover` (a cover's id, or none) and `layers` (of it left), or `null`.

The rules are written in a small set of **helpers** at the top of `01-core.js` (part 1, chapter 11, lists them): where squares are (`rowOf`, `neighbours`, `around`, `rowSquares` …), which hold what (`bareStones`, `plainAmulets`, `edgeOfFloor`, `squaresWhere` …), choosing (`pickSome`, `pickOne`, both drawing from `shuffled()`, one shuffle for every rule) and changing (`thicken`, `coverAmulet`, `breakCover`, `makeSpecial`). Every list of squares they return is in board order, so

the same random numbers choose the same squares in the web game and in Godot. Use them in a new rule; a new helper goes beside its kind, with its twin in `godot/rules/rules.gd`. `node tools/rules-test.js` checks every helper, cover, special, badge power and hardship on small boards of their own.

`Core.play(a, b, each)` is one whole move with nothing shown (swap, clear and fall until still, spread covers, shuffle a stuck board, use a move): the simulators and the parity test all play through it and the game's `attemptSwap()` and `cascade()` do the same with pictures.

`COLS` is the number of **columns** and `ROWS` the number of rows; boards are often taller than wide, so index by `sq = row * COLS + col` and never assume a square. `COLS` and `ROWS` are module-level `let`s set by `setBoardSize()` before a stop is built.

A move

`isValid` → `swap` → `clearStep` → `gravity` → `clearStep` again, until nothing clears. `clearStep` finds runs of three or more, decides which specials are made, sets off specials caught in the blast (chains included), fires the powers of cleared badges, reduces the floor under everything cleared and **returns lists of what happened**. The game layer animates those lists and never works out a rule itself, which is why a bot can play the same code headlessly.

Specials come from the shape of a match: four in a row makes a banded amulet, an L or T a ringed one (a star if one arm is four long) and five in a row a sun. Two specials swapped together combine (`COMBO`).

Badges are marks a new amulet can fall wearing, at the difficulty's `powerChance`, picked by each badge's `weight` (`pickBadge()`). When a badged amulet clears, `clearStep` calls its power from `BADGE_POWERS`, which records what it did (`targets`, extra gilding) for the game to show. Cursed badges are kept rare and switched off on Relaxed and in river events.

Building a stop: `stopOptions()`

Every board is built through `stopOptions()`, the game's and the simulators' alike, so a test plays the stop a player gets. It takes the stop, the difficulty, the board mode and size, Treasury upgrades, persistence, omens, a curse and whether badges have arrived yet and returns the options `Core` is made with: the floor plan, the moves, the badge chance. Hardships (curses and omens) change these options through their `options(o, n)` part and change the filled board through `board(core, n)`.

Board modes

`BOARD_MODES` has one entry per board: `classic` (8 wide), `grand`, `ruins` (generated ruins, `ruinsPlan()`) and `omega` (as large as the window, `omegaPlan()`). Each gives its `layout()`, how its `moves` scale and `tall` (extra moves for a tall board). Omega is `dynamic`: its size is measured from the window when a stop starts, by `omegaSize()` and anything that previews a board must measure the same way. Generated plans must pass `allMatchable(rows)`: every playable square must lie on some run of three.

Conditions

Relics, looks, seals and staging use one condition language: a block like `{"stops_won": 10, "win_on_difficulty": "Hard"}` where every entry must hold. `conditionMet(when, s)` checks it; `CONDITION_COUNTERS` contains the counted ones and the win conditions read `s.lastWin`. `needProgress()` in

`game/09-unlocks.js` gives the "7 of 9" shown on a locked look and `conditionText()` the words.

---

5. The running game

- **State** (`game/05-state.js`): `startLevel()` builds a stop, sets up the board frame, the scenery, the note beside the board and the HUD. `updateHUD()` refreshes the numbers and the gilding tube.
- **Drawing** (`game/19-draw.js`): one canvas. `frame()` updates and draws the amulets, badges and effects each animation frame. The floor and its shadows are drawn once into an offscreen canvas (`game/01-caches.js`) and copied; anything that changes `core.floor` must set `bgDirty = true`.
- **Moves** (`game/21-moves.js`): swapping, `showClear()` animating what `clearStep` returned, falling, cascades, scoring and earnings.
- **Input** (`game/22-input.js`): pointer, drag and keyboard. Keyboard play (arrows, Enter, Escape) must keep working.
- **Effects** (`game/20-effects.js`): sparks, beams, orbs, rings, flashes, popups and vibration, all plain arrays that `frame()` draws.
- **Busy**: while a move plays out, `busy` is true and input waits. `settle()` returns a promise that resolves when the board is still.

---

6. Screens and the interface

The interface has a small set of rules. Follow them and a new screen looks like the rest without new CSS.

Scrolls: `showMsg()`

Almost every screen is a scroll (`#ovMsg`), opened with `showMsg(html, actions, opts)` in `game/23-scrolls.js`. Each action is `[label, what it does]`, with an optional third part saying how it looks:

```
showMsg(html, [
    [Tplain('win.sail', { stop: next.name }), () => goNext(i + 1), { kind: 'go', sub: next.sub,
icon: iconSvg('ui', 'boat') }],
    [Tplain(placeKey(door, 'explore'), { name }), () => startChamber(door), { kind: 'card', dark:
true, icon: iconSvg('map', placeIcon(door)) }],
    [Tplain('win.map'), openMap, { kind: 'quiet', icon: iconSvg('dock', 'map') }],
], { noClose: true });
```

- `go`: the usual next step. **One per scroll**: a big stone button, full width, with a line under it (`sub`) saying where it goes.
- `card`: another way to go, smaller, with a line under it.
- `dark: true`: the way into a tomb, in torchlit brown; with `oasis: true`, blue daylight.
- `quiet`: everything else (the map, replaying, leaving): small buttons in a row at the foot. A single `quiet` beside a lone `go` shares its line instead, as tall as the big button; give it `short` words ("Leave") for phones, where its long ones won't fit.
- `exit`: a way out that changes nothing ("No trial today", "Back to the board"): full width at the very foot, after a ×. `exitButton()` makes the same button outside `showMsg` (the map's).
- `hidden: true` starts an action hidden, for a screen that swaps its big button (the stop card does, per tile).

- An action with none of these is a plain button, as in the shops.

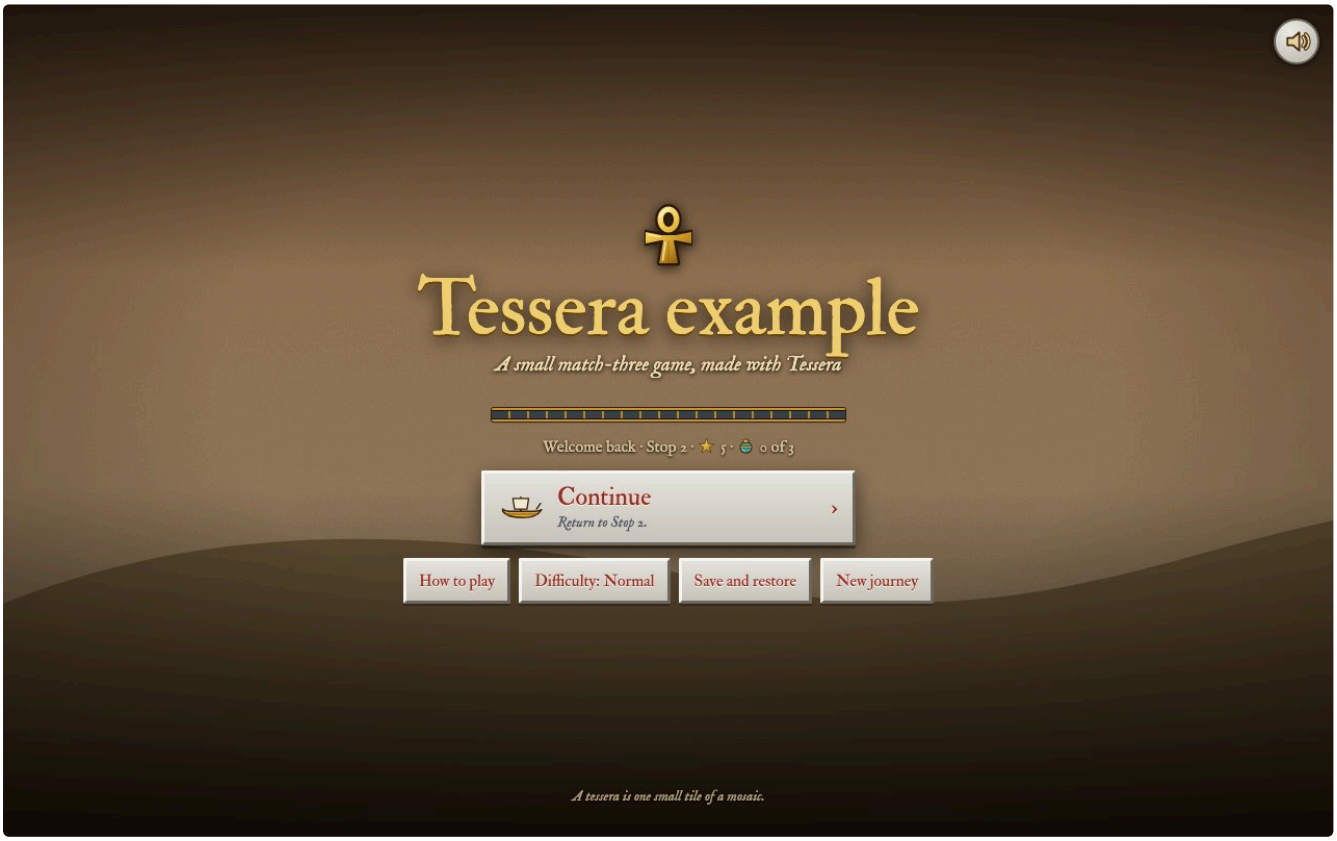
`opts.noClose` hides the × and blocks Escape, for scrolls that need a decision (winning, losing). `opts.onClose` says what closing means when it isn't "back to the board": the stop card's × goes back to the map.

The visual language

- **Buttons are stone**: a gradient in the `--stone-*` colours with a bevelled border (light top-left, dark bottom-right). **Chosen is pressed in**: the bevel reversed, a lighter face and a 2 px gold ring. Tabs, tiles, omen cards and the boon being aimed all use this.
- **Places have their own colour**: torchlit brown for a chamber in the `tomb` setting, blue daylight for one in the `oasis` setting, the stop's scenery for the title screen.
- **Dividers** are the band of the board's frame (`.title-hr`), in the game's `--band` colours.
- **No scroll bars where they can be avoided.** Lay out to fit; when text might not fit, show whole lines and a way to read the rest.
- **Scrolls are a column of rod, sheet and rod.** The rods are `flex: none`, or a long scroll squeezes them thin.
- Words go through `T()`; nothing English is written into the code.

The title screen

The title screen is its own layer, `#ovTitle`, over the scenery of the stop you're at. `openTitle()` (`game/15-title.js`) sets `body.at-title`, which hides the rest of the game so the scenery shows through. At start-up the page carries `body.booting`, which hides everything but the overlays until the title is up, so the board never flashes first. `closeOverlays()` takes `at-title` away.

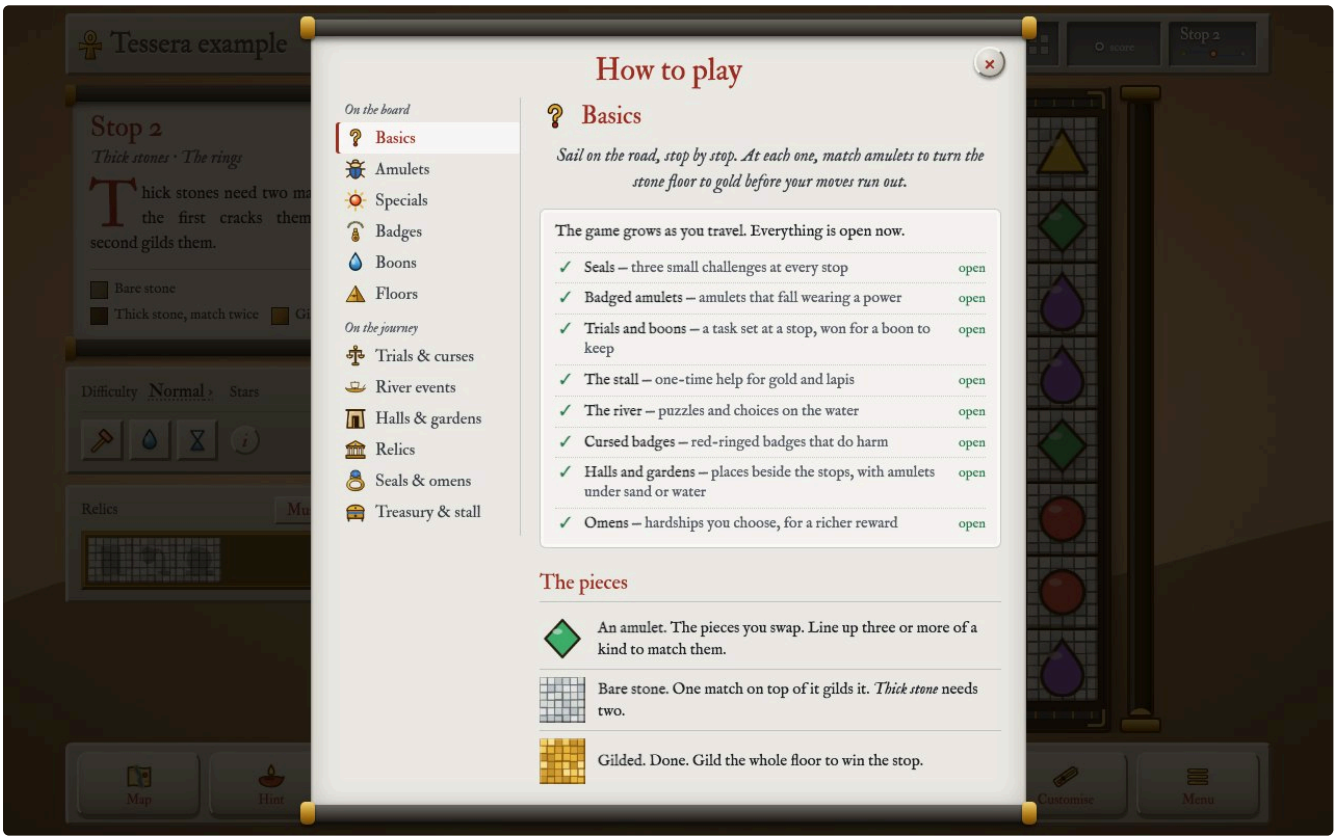


The Menu

`openMenu()` (`game/14-menu.js`): the big “Back to the board”, the band, **eight tiles** in two rows of four and the three ways out of the game (title screen, save and restore, a new journey) small at the foot. A tile carries a word of state where it helps (`Normal`, `1 of 3`, `music off`) and its full description as a tooltip. Until the stall opens on a first journey, its tile is Learning pace, so the rows stay full. A new entry should replace one, or become the ninth and tenth together.

How to play

`openHelp(tab)` (`game/25-codex.js`). The chapters are `CODEX_GROUPS`, in two groups, each with a picture in `images/icons/codex/<id>.svg` and a name in `codex.tabs`. On a wide screen they run down the margin as an index; on a narrow one (`CODEX_NARROW`, 640 px, the same as the stylesheet) How to play opens on a contents page (`openHelp('contents')`) and each chapter has the next and previous at its foot. A chapter waits for its stage on a first journey (`CODEX_STAGE`). A game’s own website can read `CODEX_GROUPS` from this file to lay out its How to play the same way.



The stop card

`openStop(i)` (`game/28-map.js`): with more than one part to show (seals, omens, a doorway), each is a tile with its gist (“1 of 3”, “2 braved”, “explored”) and only the chosen one is open. The big button follows the tile: “Set out” onto the stop, or into the doorway on its tile.

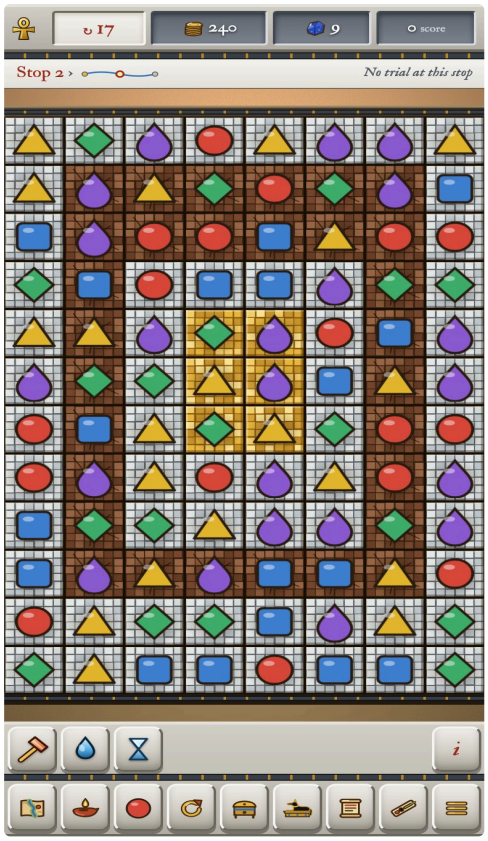
The column beside the board

On a desktop, the board has a column beside it: the stop’s note, a tablet (difficulty, stars, trial, boons) and the relics. It must fit without scroll bars:

- **The note** shows whole lines only. `fitNote()` (`game/05-state.js`) measures it; if it overflows, the key to the stones goes first, then the note is cut to whole lines and ends in “Read on”, which opens it in a scroll. A `ResizeObserver` on the tablet runs it again when the tablet grows.
- **The boons** are icons with names on a tall window and icons only when the window is shorter than 820 px. A ? after them opens How to play at Boons. On a phone they are icons only, at least 44 px, for thumbs.
- **The relics** are one row, found ones first, with a Museum button; the row opens the Museum too.



On a phone the column becomes a strip under the board and the note opens from it:



Words on screen

Every player-facing word is in `content/text.jsonc`. `T('win.title', {stop})` returns the text with `**bold**`, `*italic*` and new lines made HTML, `{stop}` filled in and `{n|stone|stones}` chosen by number; `Tplain()` gives plain text for titles and labels. In `engine/web/shell.html`, `{{hud.moves}}` is filled at build time. `plainText()` strips HTML for attributes such as `title`.

Icons

Icons are SVG files in `images/icons/<group>/`, never strings in the code. `iconSvg(group, name, attrs)` gives a whole icon; `iconArt(group, name)` its inside, with its `<defs>` moved once into the shared defs, because Firefox loses a gradient whose first copy is hidden. So gradient ids must be unique across icon files (only `relicGold` is shared on purpose). In `engine/web/shell.html`, `{{svg:dock/map}}` puts an icon in at build time.

Staging and “new”

A new player meets the game’s parts one at a time: `STAGES` and `stageOn(id)` in `game/26-stages.js`, with the stop each arrives at in `settings.jsonc`. Until its stage, a part is hidden, not just quiet: its buttons, Menu tile, How to play chapter and map marks. `announce()` shows a banner; `markNew(key)` puts a red dot on the button where something can be found until the player opens it (`clearNew(key)`). A new stage needs `title`, `text` and `what` in `text.jsonc` (`stages.<id>`).

Accessibility

Keep keyboard play working, `aria-labels` on icon buttons, focus on the first sensible button of a scroll (the title screen's sound button comes last in its markup for this reason) and respect less motion: the player's choice in Settings, or the phone's own "reduce motion" (`applyMotion()` in `00-open.js` sets `reduceMotion` for the code and the `calm` class on `<html>` for the stylesheet). In CSS write `:root.calm .thing` rather than `@media (prefers-reduced-motion)`, so the setting can override the phone.

Settings

Settings (`openSettings()`, `14-menu.js`) has five parts as tiles, each opening one: sound and music, vibration, colours, motion and effects and this device. Sound, vibration, colours and motion are in the save (`colours`, `motion`); how the board is drawn (`deviceSetting('renderer')`, kept under the edition's `device_prefix`) and fewer effects (`deviceSetting('effects')`) belong to the device and are kept in the browser (`deviceSetting()`), so a save code taken to another phone doesn't bring them. Effects on "Automatic" turn down only on a phone that reports 4 GB of memory or less, after a fifth of the frames during cascades were slow at two different stops (`watchEffects()`, `19-draw.js`).

Colour modes

The game's own colours, dark, high contrast light and high contrast dark, or following the phone (`applyColours()` sets `data-colours` on `<html>`). Every colour of the interface is a named variable in `00-colours.css` and each mode is one list there; the rest of the stylesheet writes `var(--ink)`, `var(--bevel-hi)` and so on, never a colour of its own. The names say what a colour is for, not what it is made of: `--paper` for scrolls and notes, `--ink`, `--accent` (headings, warnings, the moves), `--focus` (the keyboard's ring), `--ink-cool`, `--good`, the `--stone-*` buttons, the `--slab` bars, `--band` and `--band-edge`, `--page` behind everything, the map's `--map-*` labels. There are two files of that name: the engine's (`engine/web/css/00-colours.css`, a plain look) and a game's own (`web/css/00-colours.css`), which takes its place (chapter 13). Both set every name, so a new name goes in both. What looks the same in every mode keeps its colour where it is used: gold and sand, shadows, water, the title screen and anything on the board. So a new rule that colours part of the interface uses the names; if none fits, add one to all four lists, in both files.

7. The systems

System	Where
Trials: offered, measured, rewarded	`content/trials/`; `trialProgress()` in `game/07-trials-boons.js`, `levelWon()` in `game/24-win-lose.js`
Boons: held, aimed, used	`content/boons/`; `BOON_EFFECTS`, `useBoon()` in `game/07-trials-boons.js`
Curses and omens	`content/curses/`, `content/omens/`; `HARDSHIPS` in `01-core.js`, applied through `stopOptions()`
Seals	`"seals"` in each stop; `stampSeals()` in `game/27-seals.js`
Relics	`content/relics/`; `checkRelics()` in `game/09-unlocks.js`
River events	`content/river-events/`; `game/10-river-events.js`, on a small board (`setupSmallBoard()`)

System	Where
Chambers	`content/chambers/`; `game/11-chambers.js`. `placeKey()` picks an oasis's words, `placeIcon()` its picture, `coversNote()` the card's line about its covers
Covers	`content/covers/`; `COVERS` in `01-core.js`, broken in `clearStep()`, grown by `Core.spread()`; drawn with `coverPicture()` (`02-pictures.js`), burst and heard in `showCovers()` (`game/21-moves.js`)
The stall, the Treasury	`content/stall/`, `content/treasury/`; `game/12-stall.js`, `game/16-treasury.js`. Every purchase is logged with `logBuy()` and can be undone until the screen closes
Looks	`content/amulet-sets/` and the rest; changing one goes through `applyLook()`, which also empties the pre-scaled amulet cache. An amulet set's own pictures (`images/amulet-sets/<id>/`) are loaded into `SET_PICS` (`02-pictures.js`) and used by `skinned()` (`04-boards.js`); a set with `pixel_art` is drawn without smoothing, there and in the sprite cache (`game/01-caches.js`)

8. Sound and music

Everything is synthesised with Web Audio, unless a game brings recordings of its own (below).

- **Effects** (`game/03-sound.js`): plucked strings (Karplus-Strong), bells, gongs and stone, mostly rendered once into buffers and replayed.
- **Music** (`game/04-music.js`): generated live in the stop's key, scale and instrument. `PROGRESSIONS` are chord sequences through the scale; `MUSIC_LAYERS` the voices; `musicArrange()` brings them in and out; `musicFollow()` turns moves left into tension and gilding into brightness; `musicResolve()` ends a stop; `musicDuck()` muffles it under a scroll. Pitched effects take their notes from the chord playing (`chordTone()`).
- **Recordings** (`sounds/` and `music/` in the game's folder): the build embeds them as `data:` addresses in `PICTURES.sounds` and checks the names (an effect's name from the `cases` of `sfx()`, a stop's id or `default`). `soundBuffer()` (`game/03-sound.js`) decodes each once; `sfx()` first tries `playRecorded()`, which plays the recording through `sfxOut` (so the volume and mute apply) and otherwise, or while it's still being decoded, falls through to the made-up sound. For music, `musicFile()` picks the stop's piece or `default` and `playMusicFile()` loops it through `music.bus`, so the volume, `musicDuck()` and the fade at the end of a stop work as usual; `musicFollow()` and the chord-tuned effects don't apply to a recording. Nothing is fetched: it all comes out of the one file.
- **Loudness**: music about −22 LUFS at the default volume, some 6 dB under the effects. Measure by recording the master output. Nothing below about 50 Hz.
- **Crackle on phones** is an audio-thread problem, not a frame-rate one: see chapter 12.

9. Extending the engine

Most additions are content (Part 1). These need code.

A new boon effect

Add an entry to `BOON_EFFECTS` (`game/07-trials-boons.js`): `target` (does the player pick a square), `amount` (the default) and `use(b, sq)` (`b.n` is the amount, `sq` the chosen square). A boon that clears squares should call `cascade(null, keys, true)`: the `true` keeps the blast from advancing a trial. The build reads the names from this list. Add the effect to the comment in the example's first boon file (which is what `new.py` copies) and to the content reference. Part 1, chapter 11, walks through one line by line.

A new badge power

Two halves. The rule goes in `BADGE_POWERS` (`01-core.js`): `amount` and `fire(core, at, n)`, which acts inside `clearStep` and returns what it did. How it shows goes in `BADGE_SHOWS` (`game/21-moves.js`): its sounds, whether it buzzes and `show()` for popups, beams and orbs. Keep the rule free of browser code: the simulators run it.

A new hardship, for curses and omens

`HARDSHIPS` in `01-core.js`. `options(o, n)` changes the stop's options before it is built (moves, badge chance); `board(core, n, h)` changes the board once filled (`h` is the curse or omen; `h.cover` the cover it names). Either or both. Written with the helpers, most are a line:

```
    // n amulets, chosen at random, start under a cover
    buried_amulets: {
        stage: 'chambers',
        board: (core, n, h) => coverAmulets(core, n, h.cover),
    },
```

A new cover

No code: a file in `content/covers/` and a picture in `images/covers/` (part 1, chapter 8). What a cover can do is its fields: `layers`, `broken_by`, `matches`, `spreads`. A new *kind* of behaviour (a cover that stops the amulet falling, say) is a new field: read it in `cover()` in `engine/build.py`, act on it where covers are handled in `Core` (`typeAt()`, `isValid()`, `clearStep()`, `spread()`, `shuffle()`), in `core.gd` the same and add it to `tools/rules-test.js`.

A new trial goal

Add the name to `TRIAL_GOALS` in `engine/build.py` and measure it in `trialProgress()` (during play) and `levelWon()` (goals judged at the end).

A new condition

Counted conditions go in `CONDITION_COUNTERS` (`01-core.js`) and `COUNT_CONDITIONS` (`engine/build.py`); win conditions in `conditionMet()` and `WIN_CONDITIONS`. Its words go in `text.jsonc` under `conditions`.

A new board mode

An entry in `BOARD_MODES` with `layout()`, `moves` and `tall`, its name in `BOARD_MODES` in `engine/build.py` and in `text.jsonc` (`boards`). A generated layout must pass `allMatchable()`. Run the simulators on it.

A new special

Its pictures in `images/specials/`, their names in `SPECIAL_PICTURES` in `engine/build.py`, its power in `SPECIAL_POWERS` (`fire(core, at)`, as a badge power's), what match makes it in `clearStep`, its drawing in `drawTile()`.

A new stage

An entry in `STAGE_MARKS` (`game/26-stages.js`), its stop in `settings.jsonc` (`staging`, parsed in `settings()`), its words in `text.jsonc` (`stages.<id>`) and `stageOn('<id>')` wherever it shows.

A new setting

A field in `settings.jsonc`, parsed with a default in `settings()` in `engine/build.py`, read as `CONTENT.settings`. Tuning numbers never go in the code.

A new saved field

See chapter 10: it needs a default in `applySaveDefaults()`.

---

10. Saving

A player's progress is kept in `localStorage` under the edition's `save_key` and players move it between copies with export codes that start with its `export_prefix` (Save and restore). Both are the game's, in `edition.jsonc`, with the looks every player starts with; the engine reads them as `CONTENT.edition`. Once people play they never change. A game does well to keep a test that a save and a code from an old version still load (*Amulets of Anubis* keeps one, `tools/old-saves.mjs`). So:

- **Never rename or repurpose a saved field.** New fields get a default in `applySaveDefaults()` (`engine/src/00-open.js`). If a field's meaning changes, write a migration there.
- Progress is keyed by **stop id** and the save records the order of ids it was made with, so adding or reordering stops moves progress with its stop (`remapStops`). A game whose saves came before stops had ids lists its first order of stops in `edition.jsonc` (`stops_before_ids`); a new game leaves it out.
- Every write goes through `persist()`, in a `try`; the game runs even when storage fails.
- Try-out mode (`#try`) uses a separate save and never touches the real one.

---

11. Balance and the simulators

The simulators run `01-core.js` in Node with the checked content (`engine/tools/load-core.js`, through `build.py --content-json`) and build boards through `stopOptions()`.

```
node engine/tools/journey-sim.js 1 60          # first journeys, as a new player meets them
node engine/tools/journey-sim.js 1 60 later    # later journeys, everything on
node engine/tools/sim.js 1 classic 8 13 24     # difficulty, board, columns, rows, games
node engine/tools/events-sim.js                # river event puzzles
node engine/tools/econ-sim.js 8 13 4 1         # earnings and unlocks over whole journeys
```

The bot is greedy and never plans, so people do a little better. Targets on Normal: about 65–75% first-try wins, first and later journeys alike, on every board, early stops higher than late ones; Relaxed about 95%. At 60 journeys one stop's figure moves about ten points between runs, so judge averages. River puzzles: between about 50% and 90%. Economy: about a third of the looks after one journey.

What tunes what: each stop's `moves`, prices, event moves and the numbers in `settings.jsonc` (content); `MOVE_TUNING`, `BOARD_MODES` and `DIFFICULTY` in `01-core.js` (code).

---

12. Performance

The target is a mid-range Android phone.

- **The floor** is cached in an offscreen canvas and only changed squares are repainted after a match (`bgCells`).
- **Amulets** are pre-scaled to the square size (`tileAt`).
- **Idle frames**: 30 a second when something could move, 4 when nothing on the board moves by itself (`idleWanted()`).
- **Fewer effects** (`lowFx`): chosen in Settings, or turned on by "Automatic" only on a phone with 4 GB of memory or less that struggles during cascades at two stops. A good phone is never switched by a hitch.
- **No blurred CSS on or round the board**: Firefox repaints it every frame. Paint shading into the floor layer instead.
- **Gradients** used by repeated icons live once in the shared defs.
- **The sand tube** is a still tiled picture; only its height moves and its stream of grains shows only while it fills.
- **Canvas size** is capped at about two million pixels.
- **Audio**: phones ask for a larger buffer (`latencyHint: 'playback'`); reverbs are one channel and short; plucked strings, bells and gongs are buffers made once; music is scheduled 0.4 s ahead and nothing starts sooner than 30 ms from now (`MUSIC_AHEAD`, `MUSIC_MARGIN`, `SFX_LEAD`); "Steadier sound" asks for a fixed buffer (`sound.steady_buffer_ms`). Judge an audio change by the work per render slice: render it in an `OfflineAudioContext` with `suspend()` every 0.1 s and time each slice.
- A game can check for leaks by playing a long session and watching the page's memory (*Amulets of Anubis* keeps its own soak test).

---

13. The stylesheet

`engine/web/css/` is joined in file-name order, so the numbers are the order.

A game's own look. A game can bring stylesheet files of its own, in its `web/css/`. One with the same name as an engine file takes that file's place; one with another name is added, in the same file-name order (so `05-extra.css` comes after the engine's `05-dock.css`). *Amulets of Anubis* brings one, `web/css/00-colours.css`: every colour and texture of its papyrus, sandstone, painted band and page, in the four modes. A game with none gets the engine's plain look (light

paper, grey stone, a little gold). A game's `web/fonts.css` likewise replaces the engine's typefaces (then set `--display` and `--body` in a file of its own after `01-base.css`, such as `01-typefaces.css`) and its `images/favicon.svg` (or `.png`) is the picture in the browser's tab.

File	Styles
`00-colours.css`	every colour and texture by name and the four colour modes (a game's own replaces it)
`01-base.css`	fonts, the page, the scenery, less motion (`.calm`), what a mode changes beyond colours
`02-layout.css`	the app grid
`03-hud.css`	the top bar and coins (`.g-ico`) anywhere
`04-board.css`	the stage, the board frame and canvas, the sand tube
`05-dock.css`	the buttons along the bottom
`06-scroll-and-buttons.css`	scrolls and their rods (`.paper`), stone buttons, the drop cap
`07-side-panel.css`	the column beside the board: note, boons, relics
`08-shops-and-looks.css`	Treasury, stall, Customise
`09-codex-and-sound.css`	How to play, Customise tabs, Settings
`10-screens.css`	the ×, the title screen, the Menu, dividers
`11-journey.css`	seals, omens, the stop card's tiles
`12-notices.css`	banners, "new" dots, the try-out badge
`13-overlays.css`	scrolls, `showMsg()`'s button kinds, the map
`20-small-screens.css`	phones and short windows, last on purpose

Colours are named in `00-colours.css` (see "Colour modes"). Put a rule in the file for its part of the screen; phone rules in `20-small-screens.css` or an `@media` block beside the rule.

14. Testing

After any change to `engine/src/`:

```
python3 build.py
grep -c "https://" dist/<file>.html              # must print 0
node engine/tools/screenshots/smoke.mjs
```

The smoke test plays the built game for a moment as a new player and as one who is part-way through a journey, on a desktop and a phone: moves, every screen, every tab and chapter. It fails on any page error. The simulators only run `01-core.js`, so they can't catch a mistake in `engine/src/game/`.

Also: `node engine/tools/screenshots/edge.mjs` plays the awkward paths (losing and leaving by the map, winning and coming back, taps in the middle of a move, turning the phone, broken saves) and reports dead ends; `node engine/tools/screenshots/check-looks.mjs` checks every amulet set for amulets too close in colour; `node`

`engine/tools/screenshots/pdf.mjs` prints an HTML page to a PDF. The browser tests need Playwright, installed once: `cd engine/tools/screenshots && npm install && npx playwright install chromium`. Check layout at 390 × 844 and 360 × 780 as well as on a desktop: the page must never scroll.

---

15. Packaging

Tessera builds the web game: one file that plays in any browser. That file is the thing to keep. Everything else is a wrapper round that file and belongs to the game, not the engine:

- **An installable web app** is the file served as `index.html` beside a `manifest.webmanifest` and a small service worker that caches it; change the service worker's cache name for each release, so installed copies fetch the new one.
- **An Android app** is a WebView showing the file from the app's assets, with storage and the back button wired up. The engine handles Back through the page's history and keeps clear of a notch through CSS variables (`--cut-top` and the rest, falling back to the browser's safe area), so a wrapper only has to report the notch.
- **Desktop apps** are the file in a frozen browser engine such as Electron, with the network blocked.

Amulets of Anubis, the first game made with Tessera, has all three (`platforms/` and `build.sh` there), with its app id and name written in. Until wrappers read `edition.jsonc`, a second game copies them and changes those.

---

16. House style

- **Nothing of one game in the engine.** A name, a word, a number, a picture or a colour that belongs to one game goes in that game's files and the engine reads it from there; the build checks it.
- **British English** in the engine's comments, messages and docs.
- **The engine's own words** for its pieces: stops, stones, floors, gilded, amulets, specials, badges, boons, trials, curses, relics. What a game calls them on screen is its own business, in `text.jsonc`.
- **Watch the engine's size.** Prefer a content file to new code and say how much code a change adds.
- **Every file in `src/` starts with a note** (what it is for, its main functions, what it changes in the save); keep it up to date.

---

17. The Godot version

`engine/godot/` is a version of the engine made with Godot, a free game engine, which draws its whole interface itself. It uses the same content (read through `build.py --content-json`, never from the content files directly), the same pictures and the same rules. `src/01-core.js` stays the one true version of the rules and the Godot port follows it name for name. A parity test plays seeded games in both and compares them, move for move.

Today, it plays the journey of any Tessera game: the title screen, the map, the stops one after another, winning and losing, the Menu and saves that move to and from the web game. Trials, boons, the shops, events, chambers, looks, How to play and sound are being worked on (sound is finished on a branch of its own). `godot/README.md` explains what's done so far and how to run it.

Appendix A: Content reference

The place to look up the exact name of a setting. If you're just getting started, part 1 of the manual is the better place to begin, since it explains all of this step by step.

This reference lists every setting of every kind of content file, which can be found in your game's `content/` folder. Each file describes one thing, such as a stop or a boon. When you build, the build (`engine/build.py`) reads every one of these files and checks each setting. If something is wrong, it stops and tells you which file, which line and what to do about it. In that case, it doesn't write anything.

The examples below are shortened to what matters. The example game in `example/content/` has a working file of almost every kind. The first file of each kind also explains every one of its settings in comments. Those first files are also what `new.sh` copies when you make something new.

The rules for every file

- **One thing per file.** A file contains a single `{ ... }` block.
- **The file name decides the order.** The number at the front sets the order of the things of that kind: `03-stop-3.jsonc` comes after `02-stop-2.jsonc`. To reorder things, simply renumber the files.
- **An `id` is permanent.** Saves remember things by it. Use lower case letters, numbers and hyphens, starting with a letter, like `"old-harbour"`. Don't change an id once people have played your game.
- **Kinds of values.** Text goes in double quotes, numbers go without them, yes or no is `true` or `false` and lists go in square brackets. Long text can also be written as a list of lines, which the build joins with spaces: `["the first part", "the second part"]`.
- **Comments** start with `//` and are fine anywhere, so each file can explain its own settings. That's also why the files end in `.jsonc`, "JSON with comments": editors then know that comments are allowed. The build forgives a comma right before a closing bracket, but some editors don't, so it's best to leave it out.
- **Colours** are written as `#rrggbb` (or `#rgb`), or as `rgba(…)` or `hsla(…)` for see-through colours.
- **A setting the build doesn't know** is ignored. The build tells you about it, usually along with the name it thinks you meant.

The floor plan

Used by stops, floor shapes, event puzzles and chambers.

A floor plan is a list of eight lines with eight characters each. Every character is one square:

Character	Square
`.`	a gap: there is no square here
`0`	stone that is already gilded, so there's nothing to do
`1`	bare stone: one match gilds it

Character	Square
`2`	thick stone: it takes two matches
a cover's letters	an amulet under that cover, on bare stone (the first letter) or on thick stone (the second). Each cover in `content/covers/` has two letters of its own. In the example, these are `s`/`S` for sand, `w`/`W` for water, `a`/`A` for amber, `n`/`N` for a net and `v`/`V` for ivy.

```
"floor_plan": [
  "00000000",
  "01111110",
  "01111110",
  "01122110",
  "01122110",
  "01111110",
  "01111110",
  "00000000"
]
```

What the build checks. Every square that isn't a gap has to sit on a row of three squares, across or down, or it could never be matched. There has to be some stone to gild at all: a stop's plan without a single `1` or `2` is refused. Where a plan has covers, every patch of covers that doesn't match has to touch a square that starts clear, so the player can reach it. At least *16* squares have to start clear.

A covered amulet can't be moved. Unless its cover says otherwise (the cover's `matches`), it isn't part of any match either. Once it's free, it's an ordinary amulet again and its stone is gilded when it's matched.

Conditions

Used by relics (`found_when`*) and by the looks (*`unlocked_by`*).*

A condition block is a group of conditions and **every one of them has to be met**. `null` means "right from the start". A number means "at least this many".

Counted over all of the player's journeys:

Condition	What it counts
`stars`	stars earned
`three_star_stops`	stops with three stars
`stops_gilded`	stops won at least once
`stops_won`	stops won, all in all
`hard_wins`	stops won on the third or fourth difficulty
`trials_finished`	trials finished
`river_events`	events played
`relics_found`	relics in the Museum
`suns_forged`	suns made

Condition	What it counts
`best_cascade`	the longest cascade so far
`thick_stones_cracked`	thick stones gilded
`gold_earned`	gold earned
`gold_held`	gold the player has right now
`lapis_held`	lapis the player has right now
`win_streak`	wins in a row right now
`journeys`	journeys begun
`seals_stamped`	seals stamped, at all stops
`omens_braved`	the most omens braved at each stop, added up
`chambers_explored`	chambers explored

About the stop that was just won (so a relic can be found the moment the player wins a stop that meets these):

Condition	Its value
`win_moves_to_spare`	the number of moves left
`win_on_difficulty`	a difficulty's name, such as `"Normal"`
`win_at_stop`	a stop's id
`win_without_boons`	`true`
`win_on_board`	a kind of board: `classic`, `grand`, `ruins` or `omega`
`win_two_specials_at_once`	`true`
`win_after_failures`	the number of earlier failures at this stop
`win_with_omens`	the number of omens braved
`win_suns_forged`	suns made during the stop
`win_best_cascade`	the longest cascade during the stop
`win_specials_made`	special amulets made during the stop

There is one more: `relic`, a relic's id, so that finding one relic can unlock the next.

```
"found_when": { "stops_won": 1 }
"found_when": { "win_at_stop": "stop-2", "win_moves_to_spare": 5 }
"unlocked_by": { "stars": 15 }
```

Rewards

Used by relics, events and chambers.

A reward is a group of any of these: `gold` and `lapis` (whole numbers, 1 or more) and `boon` (a boon's id, or `"random"`).

```
"reward": { "gold": 45 }
"reward": { "lapis": 25 }
"reward": { "boon": "hammer" }
```

Stops (`content/stops/`)

A stop is one place on the journey, with its own floor, amulets, scenery and music. The stops come one after the other, in the order of their file names.

Setting	Needed?	What it is
`id`	yes	its internal name, which never changes
`name`	yes	the stop's name
`subtitle`	no	a short line under the name
`history`	no	the note beside the board
`moves`	yes	how many moves the player gets, before the difficulty and the size of the screen are taken into account (5 to 120)
`amulets`	yes	the amulets that fall here: 4 to 6 of them, each one only once (see below)
`floor_plan`	yes	the stop's own floor plan
`shared_floor_shapes`	no	a list of floor shapes this stop sometimes uses instead, by their ids
`stone_colour`	no	the colour of its bare stone (`#a8977a` if left out)
`music`	no	`{key, scale, instrument}` (see below)
`scenery`	no	another stop's picture in `images/backdrops/` to use instead of its own
`board_backing`	no	another stop's picture in `images/boards/` to use instead of its own
`frame_colours`	no	exactly 3 colours for the board's frame: `[face, shade, edge]`
`map_position`	yes	`{x, y, label}`: where it sits on the map (see below)
`seals`	no	three small challenges: `[{"text": "...", "when": {conditions}}, ...]`, each stamped the first time a win here meets its `when`. Keep their order once people play, since saves remember seals by their position.

`amulets` is either one list of 4 to 6 amulet ids or a list of such lists, like `[["round", "square", "triangle", "diamond"], ["round", "drop", "star", "diamond"]]`. In that case, the game picks one of the lists at random each time the stop starts (restarts included).

`music`: `key` is a note, like `"D3"` (or a frequency from 40 to 1000 Hz). `scale` is one of `hijaz`, `double harmonic`, `dorian`, `minor`, `major`, `mixolydian`, `lydian`, `phrygian`, `major pentatonic` or `minor pentatonic`, or a list of semitone steps of your own. `instrument` is one of `harp`, `lyre`, `oud`, `flute` or `bell`.

Pictures. A stop's scenery is `images/backdrops/<id>.svg` and what shows through the gaps in its floor is `images/boards/<id>.svg` (or a `.png` or `.jpg` of the same name). The build stops if one of them is missing. `scenery` and `board_backing` name a different picture instead, without the ending: `"scenery": "stop-1"`.

`map_position`: `x` and `y` are measured in the map picture's own units, which are set by the `viewBox` of `images/icons/map/background.svg`. `label` is `"right"` or `"left"`: which side of the marker the stop's name is shown on.

Floor shapes (`content/floor-shapes/`)

A floor shape is a floor plan that several stops can share. A stop names the ones it may use in its `shared_floor_shapes` and then, now and then, it's played on one of them instead of its own `floor_plan`. That's a nice way to give a stop some variety without drawing a new stop.

Setting	Needed?	What it is
`id`	yes	its internal name, which stops use in `shared_floor_shapes`
`name`	yes	its name
`floor_plan`	yes	the floor plan (see "The floor plan" above)
`move_multiplier`	no	how many moves a stop gets on this shape, compared to its own floor: 1.2 means a fifth more. From 0.5 to 3; 1 if left out.

Amulets (`content/amulets/`)

An amulet is one of the pieces the player swaps. This file contains what How to play says about it. Its picture is `images/amulets/<id>.svg` (or `.png`).

Setting	Needed?	What it is
`id`	yes	its internal name, the same as its picture's name
`name`	yes	its name
`plural`	yes	its name for more than one, which trials use ("clear 14 round gems")
`meaning`	no	what it means, in How to play

Trials (`content/trials/`)

A trial is a task that can be offered at the start of a stop. If the player finishes it, they win its boon.

Setting	Needed?	What it is
`id`	yes	its internal name
`goal`	yes	what the player has to do (see below)

Setting	Needed?	What it is
`target`	yes	how many (1 or more)
`text`	yes	the task in words; `{target}` is filled in with the number and `{amulets}` with the amulet's name
`progress_label`	no	the word beside the count (`"Progress"` if left out)
`boon`	yes	the id of the boon it gives
`only_if_thick_stones`	no	only offer it if the floor has at least this many thick stones
`within_moves`	no	for `gild_half_quickly`: the number of moves the player has to do it in

The goals are `clear_amulets`, `make_specials`, `cascade`, `gild_in_one_move`, `clear_in_one_move`, `moves_to_spare`, `make_suns`, `make_bands`, `crack_thick`, `gild_half_quickly`, `no_boons` and `combine_specials`. A `clear_amulets` trial should have `{amulets}` in its text, so it names the amulet.

Boons (`content/boons/`)

A boon is a power the player keeps and uses whenever they like. Players keep boons by their id.

Setting	Needed?	What it is
`id`	yes	its internal name; saves keep boons by it, so don't change it
`short`	yes	the word on its button (about 8 letters)
`name`	yes	its full name, like "Mason's chisel"
`description`	yes	what it does; `{n}` is the amount
`effect`	yes	what it does (see below)
`amount`	no	the number the effect uses (each effect has a default)
`popup`	no	the words shown on the board when it's used; `{n}` is the amount
`popup_when_nothing`	no	the words shown instead when there was nothing for it to do
`random_reward`	no	`true`: it can be the boon a `"random"` reward gives
`icon`	no	the name of another icon in `images/icons/boons/` to use instead of its own

The effects (★ means the player chooses a square first; the number is the default amount):

Effect	What it does
`extra_moves`	n more moves (6)
`gild_stones`	n stones, chosen at random, lose a layer (8)
`thin_thick_stones`	every thick or cracked stone loses a layer
`cord_row` ★	every stone in the chosen row loses a layer
`shatter_one` ★	shatters the chosen amulet and gilds the stone under it

Effect	What it does
`shatter_square` ★	shatters an n by n square around the chosen amulet (5; use an odd number)
`make_banded` ★	the chosen amulet becomes a banded amulet
`make_sun` ★	the chosen amulet becomes a sun
`make_ringed`	n amulets, chosen at random, become ringed amulets (3)
`shuffle`	the amulets are shuffled and the player gets n more moves (2)

Badges (`content/badges/`)

A badge is a mark some amulets wear when they fall in. When the amulet is matched, the badge's power is used. Its picture is `images/badges/<id>.svg` (*96* by *96*, with the badge as a circle with a radius of *24* in the middle); the build won't accept a badge without one.

Setting	Needed?	What it is
`id`	yes	its internal name, the same as its picture's name
`name`	yes	its name, like "Gilding badge"
`looks`	yes	a few words on how it looks, shown after its name in How to play, like "gold sparkle"
`text`	yes	what it does, in How to play; `{n}` is the amount
`effect`	yes	its power (see below)
`amount`	no	the number the power uses (each power has a default)
`popup`	yes	the word shown on the board when it's used; `{n}` is the amount
`popup_when_nothing`	no	the word shown instead when it couldn't do anything
`cursed`	no	`true`: it wears a red ring and never falls on the easiest difficulty or in events (`false` if left out)
`weight`	yes	how often it turns up compared to the other badges; 0 switches it off
`colour`	yes	the colour of the glow around an amulet wearing it, like `"#ffd65a"`

The powers (the number is the default amount):

Power	What it does
`gild_stones`	gilds n bare stones anywhere (4)
`gild_around`	gilds the stones in the nine squares around it
`gild_wide`	gilds the stones up to n squares away on every side (2)
`gild_row_and_column`	gilds the stones in its row and its column, without clearing anything
`row_and_column`	clears its whole row and its whole column
`clear_around`	clears everything up to n squares away on every side (1)

Power	What it does
`clear_its_kind`	clears every amulet of its own kind
`break_covers`	breaks one layer of every cover on the board
`extra_moves`	n more moves (3)
`give_lapis`	n lapis (3)
`lose_moves`	n moves lost, but never the last one (2)
`ungild_stones`	n gilded stones turn bare again (3)
`thicken_stones`	n bare stones turn thick (3)
`cover_amulets`	n plain amulets go under the game's first cover (2)

The last four are meant for cursed badges.

Covers (`content/covers/`)

A cover is something that sits atop an amulet and keeps it in place, such as sand, water, amber, a net or ivy. A covered amulet can't be swapped, doesn't set off its special or badge and stays where it is when the board is shuffled. Its picture is `images/covers/<id>.svg`, drawn over the amulet on a canvas of 192 by 192 (the amulet's square goes from 32 to 160). A cover can also have `<id>-1.svg`, `<id>-2.svg` and so on, for when it looks different with that many layers left. The first cover (by file name) is the one curses, omens and the `cover_amulets` badge power use, unless they're told otherwise. Part 1, chapter 8, "A new cover", explains all of this with a worked example.

Setting	Needed?	What it is
`id`	yes	its internal name, the same as its picture's name
`name`	yes	its name in How to play, like "Buried in sand"
`text`	yes	what it means, in How to play
`popup`	yes	shown when the player tries to move a covered amulet
`chamber_note`	no	the line on the card of a chamber whose floor has it (the popup if left out)
`letters`	yes	its two letters for floor plans, on bare stone and on thick stone, like `["s", "S"]`. No other cover may use them and neither may be `.` or a digit.
`layers`	no	how many hits it takes to break, from 1 (the default) to 9
`broken_by`	no	`"beside"` (the default): a clear on it or next to it is a hit. `"on"`: only a clear of its own square is.
`matches`	no	`true`: the amulet under it still counts in a match where it lies; the match takes off a layer and the amulet stays. `false` (the default): it's stuck until the cover is gone.
`spreads`	no	`true`: after a move that broke none of it, it grows over one plain amulet next to it. `false` by default.
`sound`	no	the sound it makes when it breaks: `sand` (the default), `splash`, `crack`, `stone`, `gild`, `blessing`, `create` or `land`

Setting	Needed?	What it is
`burst`	no	the two colours of the specks it bursts into, like `["#e8c98a", "#b88a48"]`
`amulet_on_show`	no	the amulet How to play draws under it (the first amulet if left out)

A cover loses at most one layer in each step of clearing, however many matches touch it at once.

Curses (`content/curses/`)

A curse is what failing a trial costs the player, at the next stop only. The trial's card names its curse before the player accepts it.

Setting	Needed?	What it is
`id`	yes	its internal name; a save that carries the curse remembers it
`name`	yes	its name, like "Short road"
`text`	yes	what it does, with `{n}` for its strength: "{n} bare stones turn thick". Leave out "at the next stop", since the game adds that itself.
`text_at_full_strength`	no	used instead of `text` when the strength is 100
`effect`	yes	what it does (see below)
`strength`	yes	how strong it is on each difficulty, by the difficulties' names in `settings.jsonc`, like `{"Relaxed": 0, "Normal": 3, "Hard": 5, "Hardest": 7}`. 0 means it never happens on that difficulty; keep the easiest one at 0. At most 100.
`cover`	no	for the effects that cover amulets: which cover to use, by its id (the first cover if left out)

The effects are called **hardships** and the omens use the same list:

Hardship	What it does
`fewer_moves`	n fewer moves
`fewer_moves_percent`	n per cent fewer moves
`thick_stones`	n bare stones, chosen at random, start thick
`thick_stones_share`	one bare stone in n starts thick
`thick_edges`	n bare stones along the edge of the floor start thick
`buried_amulets`	n amulets start under a cover, as in a chamber
`covered_edges`	n amulets along the edge of the floor start under a cover
`fewer_badges`	badged amulets fall n per cent less often (100 means none at all)
`more_cursed_badges`	cursed badges fall, n times as often
`no_boons`	no boons can be used

Omens (`content/omens/`)

An omen is a hardship a player may choose to face at a stop they've already gilded, in exchange for a bigger reward (how much bigger is `omens.reward_percent_each` in `settings.jsonc`). It lasts for that stop and its restarts. Saves remember omens by their id.

Setting	Needed?	What it is
`id`	yes	its internal name; don't change it once people play
`name`	yes	its name, like "Short road"
`text`	yes	what it does; `{n}` is the amount
`effect`	yes	one of the hardships above
`amount`	yes (but not for `no_boons`)	how strong it is, from 1 to 100
`cover`	no	for the hardships that cover amulets: which cover to use

Relics (`content/relics/`)

A relic is a lasting reward for something the player has done. It's found once and then kept in the Museum for good.

Setting	Needed?	What it is
`id`	yes	its internal name
`name`	yes	the relic's name
`description`	yes	what it is, or how it's found
`found_when`	yes	a condition block
`reward`	no	a reward
`icon`	no	an icon in `images/icons/relics/`, by its name. A picture `images/icons/relics/<id>.svg` (or `images/relics/<id>.png`) is used instead if there is one.

Events (`content/river-events/`)

An event is something that can happen between two stops. There are two kinds: a puzzle and a choice. (The folder is called `river-events` because the engine began as a game about a river journey.)

Every event has these settings:

Setting	Needed?	What it is
`id`	yes	its internal name
`kind`	yes	`"puzzle"` or `"choice"`

Setting	Needed?	What it is
`title`	yes	its title on the scroll
`text`	yes	what happens, in a sentence or two
`weight`	no	how likely it is compared to the other events, from 0 to 100 (1 if left out)

How often events happen at all is `river_event_percent` in `settings.jsonc`.

A puzzle is a short game on a little board of its own:

Setting	Needed?	What it is
`goal`	yes	`{"type": "gild"}`, `{"type": "collect", "amulet": "…", "count": n}` or `{"type": "score", "points": n}`
`amulet_types`	yes	how many kinds of amulet fall (4 to 6)
`amulets`	no	which amulets. A `collect` goal needs this and the amulet to collect has to be among the first `amulet_types` of them.
`moves`	yes	how many moves the player gets (3 to 80)
`floor_plan`	yes	its board
`reward`	yes	what winning the puzzle gives (see "Rewards" above)
`by_lamplight`	no	`true` to play the board dim and small

A choice offers the player a small decision:

Setting	Needed?	What it is
`choices`	yes	a list of 1 to 5 choices, each `{label, cost?, give?, take_a_boon?, gamble?}`

`cost` and `give` are rewards: a choice can cost gold or lapis and give gold, lapis or a boon. `take_a_boon` trades away one of the player's boons. `gamble` leaves the outcome to chance. It's a good idea to always include a choice that simply lets the player carry on.

On the scroll, each choice that does something becomes a card, much like a trial's card. Its `label` is the heading, followed by "You pay:" and "You get:" lines made from its `cost`, `give`, `take_a_boon` and `gamble`. So keep the `label` a short action ("Pay for the shortcut", "Join the dance") and leave the price out of it. A choice that does nothing becomes the way out, below the cards. A card the player can't afford stays on the scroll, dimmed, with the reason.

Chambers (`content/chambers/`)

A chamber is an extra place beside a stop, with a small board of its own. It opens once that stop has been gilded (and once the "chambers" stage has arrived on a first journey; see `settings.jsonc`). The player finds it on the win screen, on the stop's card and as a doorway beside the stop on the map. The goal is always to gild the floor.

Its `setting` is either `"tomb"` (the default), which is dim and lit by torches, or `"oasis"`, which is in daylight. (These are the engine's names; what your game calls such places is up to its `text.jsonc`.) The two only differ in how they look and in one more thing: a tomb wakes a little more each time the player goes back in. Cursed badges turn up, more likely

with every return and its reward for returning grows with them (`"returning"` in `settings.jsonc`; never on the easiest difficulty and not before curses have arrived on a first journey). An oasis stays calm.

Its floor plan may use any cover's letters and its card explains every cover on its floor (with the cover's `chamber_note`). Its `reward` is paid the first time; every later visit pays its `return_reward`.

Setting	Needed?	What it is
`id`	yes	its internal name (saves remember explored chambers by it)
`at`	yes	the id of the stop it's beside; a stop can only have one chamber
`setting`	no	`"tomb"` (the default) or `"oasis"`
`scenery`	no	a picture in `images/backdrops/` to show behind the board (several chambers can share one). If left out, `images/backdrops/<id>` is used, or else the stop's own scenery.
`name`	yes	its name, as a title ("The small hall")
`text`	yes	what the place is, in two or three sentences
`moves`	yes	how many moves the player gets (3 to 80)
`floor_plan`	yes	its floor, which may use any cover's letters
`reward`	yes	a reward, paid the first time
`return_reward`	no	the gold and lapis paid on a later visit. In a tomb, this is the first return's; it grows with each return after that. If left out, it's half the gold and lapis of `reward`, or else 50 gold.
`amulet_types`	no	4 to 6 (5 if left out)
`amulets`	no	its own list of amulets (the stop's if left out), or a list of such lists, as for a stop

It uses the stop's amulets and music unless it has its own `amulets`. A tomb darkens its scenery and muffles the music, as if you were hearing it through the walls.

Checking a chamber. Type `node engine/tools/events-sim.js`. On a first visit, the bot should win it between about half and nine times in ten (50 to 90 per cent). The bot also plays each tomb as a first and a fourth return, which should come out harder.

The stall (`content/stall/`)

Something the stall sells: bought for one stop and used once (or kept until it's used, for a second wind or a boon).

Setting	Needed?	What it is
`id`	yes	its internal name
`name`	yes	its name
`description`	yes	what it does
`icon`	no	a picture in `images/icons/stall/`, without the `.svg`. Something that gives one boon shows the boon's own picture if left out.
`currency`	yes	`gold` or `lapis`

Setting	Needed?	What it is
`price`	yes	what it costs (it gets dearer by `stall_price_rise_percent` each time it's bought again at the same stop)
`gives`	yes	exactly one of the four below

`gives` is one of: `{"moves": n}` (more moves, right now), `{"reshuffle": true}` (a reshuffle, right now), `{"second_wind": n}` (another try after running out of moves, kept until it's needed) or `{"boon": [ … ]}` (a boon for the player's hand, by its id or `"random"`).

The Treasury (`content/treasury/`)

A lasting upgrade: bought one level at a time and kept for good.

Setting	Needed?	What it is
`id`	yes	its internal name
`name`	yes	its name
`description`	yes	what it does
`currency`	yes	`gold` or `lapis`
`prices`	yes	one price per level, as a list of whole numbers
`effect`	yes	what it changes (see below)
`amount_per_level`	yes	how much each level adds

The effects are `extra_moves` (more moves at every stop), `badge_chance` (added to how often badges fall), `trial_chance` (added to how often trials are offered), `win_gold` and `win_lapis` (paid for every stop won). Upgrades with the same effect add up.

The looks: amulet sets, floor sets, frames and sparkles

A look only changes how the game looks. All four kinds have the same basic settings, plus a few of their own. Players unlock looks by playing, or buy the ones that have a price.

Setting	Needed?	What it is
`id`	yes	its internal name
`name`	yes	its name
`description`	yes	what it looks like
`unlocked_by`	no	a condition block; `null` or leaving it out means "from the start"
`how_to_unlock`	no	the words on its card while it's still locked
`currency`	with a price	what it's bought with: `gold` or `lapis`

Setting	Needed?	What it is
`price`	with a currency	how much it costs in Customise. With no `unlocked_by` it can only be bought; with one, it can be earned or bought, whichever comes first

Every player starts with one look of each kind. These are the ones named in `edition.jsonc` (`starting_looks`), or otherwise the first file of each kind. **Never delete a starting look or change its id**, since saves rely on them.

Amulet sets (`content/amulet-sets/`)

The same amulet pictures, recoloured, or pictures of the set's own. Besides the settings every look has:

Setting	Needed?	What it is
`colour_changes`	no	the colour changes below, applied in the order you write them
`css_filter`	no	a ready-made CSS filter instead (a set has one or the other, not both)
`glow`	no	a colour for a soft glow around each amulet
`tint`	no	a colour laid over the amulets
`tint_strength`	no	how strong the tint is, from 0 to 1 (0.25 if left out)
`pixel_art`	no	`true`: the set's pictures are pixel art and are drawn with crisp square pixels, never smoothed

Pictures of its own. A set can bring its own pictures in `images/amulet-sets/<its id>/`, named after the amulets (`round.png`, `square.png` and so on). They're drawn instead of the usual ones and any amulet without one keeps its usual picture. Small pixel-art pictures (say 16 by 16) look their best with `"pixel_art": true`.

The colour changes are `hue_shift` (in degrees, from −360 to 360), `saturation`, `brightness`, `contrast`, `sepia` and `greyscale`. Keep the amulets easy to tell apart: shift the colours rather than taking them away, or every amulet ends up looking the same.

Floor sets (`content/floor-sets/`)

The squares under the amulets. The file gives the look its name, description and unlock; its pictures can be found in the folder `images/floors/<id>/`: `bare`, `thick` and `gilded` (or `gilded-1` to `gilded-4`, which are mixed across a golden floor). With `"uses_each_stops_own_stone": true`, the see-through `bare` and `thick` pictures are laid over each stop's own `stone_colour`.

Setting	Needed?	What it is
`uses_each_stops_own_stone`	no	`true`: the see-through `bare` and `thick` pictures are laid over each stop's own `stone_colour`

Frames (`content/frames/`)

The board's frame.

Setting	Needed?	What it is
`colours`	no	`{face, shade, edge}`; `null` or leaving it out lets each stop keep its own `frame_colours`

Sparkles (`content/sparkles/`)

The sparks when a stone is gilded.

Setting	Needed?	What it is
`colours`	yes	a list of exactly two colours

settings.jsonc

The numbers for the whole game. Every single one is optional: if you leave one out, the game uses the default shown here. The file itself has a comment beside each of them, too.

Setting	Default	What it is
`stops_open_at_start`	1	how many stops are open before any has been won
`staging.learning_pace`	"one at a time"	how new players start; or "everything now"
`staging.seals` … `staging.curses`	2, 3, 4, 5, 6, 8	the stop at which each part arrives on a first journey (seals, badges, trials, the stall, events, curses)
`staging.omens`	"after the journey"	or a stop's number
`staging.extra_moves_before_badges_percent`	50	until badges arrive on a first journey, each stop gives this much more of its moves (since the stops are balanced with badges in play)
`stars.three_stars_moves_left_percent`	30	how many of the moves have to be left for three stars
`stars.two_stars_moves_left_percent`	15	… and for two
`difficulty.<Relaxed/Normal/Hard/Hardest>`	140/100/85/72, 4/2.2/1.5/0.8, 6/9/15/0	`moves_percent`, `badge_chance_percent` and `hint_after_seconds` (0 means no hints). The four groups, from easiest to hardest, also name the difficulties (a game that names none gets Relaxed, Normal, Hard and Hardest). Curses, conditions and `text.jsonc` use these names, while players read `difficulty_names`. Saves keep a difficulty by its place, so never rename one.
`river_channel.water_edge`, `.water_middle`, `.ripples`, `.reeds`	"#1d4f73", "#2f78a6", "rgba(220,240,255,.35)", "none"	the colours of the water running down an Omega board: at its banks, in the middle, its ripples and the reeds along its banks ("none" for no reeds)

Setting	Default	What it is
`river_event_percent`	30	how often moving on to the next stop is interrupted by an event
`trial_offer_percent`	35	how often a trial is offered when a stop begins
`badges.most_good_badges_at_once`	2	how many good badges can be on the board at once
`omens.reward_percent_each`	30	extra gold and lapis for each omen braved
`seals.lapis_each`	6	lapis for each seal stamped
`stall_price_rise_percent`	30	how much dearer a stall item gets each time it's bought again at the same stop
`persistence.extra_moves_per_failure`	2	moves added for each failure in a row
`persistence.failures_that_count`	3	up to this many failures add moves
`persistence.boon_after_failures`	2	a free boon after this many failures in a row
`earnings.stones_per_gold`	2	one gold for every this many stones gilded
`earnings.specials_per_lapis`	2	one lapis for every this many specials made
`earnings.gold_for_winning`	25	gold for gilding a stop
`earnings.gold_per_spare_move`	2	plus this much for each move left over
`earnings.lapis_for_winning`	1	lapis for gilding a stop
`returning.badge_chance_percent`	5	badges on the board when the player goes back into a chamber they've explored
`returning.curse_multiplier`	3	how much likelier the cursed badges are than their weight, on the first return
`returning.each_return`, `.most`	2, 12	what each further return adds to that, up to `most`
`returning.reward_percent_each_return`	50	how much more of a chamber's `return_reward` each return pays, until the curses stop growing
`amulets_on_show.button`, `.how_to_play`	the first amulet	the amulet shown on the Amulets button and in the floor previews, then the one in How to play
`amulets_on_show.previews`	the first four amulets	four amulets to show a look with where a stop has fewer of its own

Setting	Default	What it is
`sound.steady_buffer_ms`	80	the sound buffer used when a player turns on "Steadier sound" (Menu → Settings → Sound and music). Longer is steadier, but sounds come a little later.

Pictures and icons (`images/`)

Every picture is a file of its own. Part 1 of the manual (chapter 6, "Pictures") lists the folders, the names and the sizes. `images/favicon.svg` is the picture in the browser's tab.

The map is the size of `images/icons/map/background.svg` (its `viewBox`) and each stop's `map_position` is a place on it. Icons (the folders in `images/icons/`: `dock`, `boons`, `relics`, `menu`, `codex`, `stall`, `ui` and `map`) have to be SVG files. Keep each one's `viewBox` and give gradients ids that no other icon uses, since the game shares them across the whole page. A relic uses `images/icons/relics/<its id>.svg` if there is one and otherwise the icon named by its `"icon"`; a picture in `images/relics/` is used before either. The build complains about a file that isn't one `<svg viewBox="…">` drawing.

Sounds and music (`sounds/`, `music/`)

Recordings a game brings instead of the sounds and music the engine makes up. Both folders sit in the game's folder, next to `images/`. Both are optional.

File	Plays
`sounds/<name>.mp3`	instead of the made-up sound of that name (the names are in the appendix, "Sounds")
`music/<stop id>.mp3`	over and over at that stop, instead of the made-up music
`music/default.mp3`	at every stop without a piece of its own

`.wav`, `.ogg` and `.m4a` work too. The recordings go inside the game file, so keep them small. A name the build doesn't know is left out and the build says so, with the name it thinks you meant.

text.jsonc

Every word on screen, grouped by screen (`hud`, `dock`, `title`, `menu`, `stages`, `win`, `lose`, `shop`, `stall`, `treasury`, `customise`, `saves`, `audio`, `map`, `journey`, `trial`, `event`, `popup`, `codex`, `conditions`, `boards`, `boons`, `omens`, `curses` and so on). The keys on the left stay as they are; the text on the right is yours.

In the text, you can use `**bold**`, `*italic*`, `\n` for a new line, `{name}` for something the game fills in and `{n|one|many}` for a word that depends on a number. If the game asks for a key that the file doesn't have, the build stops and names it.

The group `names` gives the word a name may start with (`"article": "The "`) and how it's written in the middle of a sentence (`"article_mid_sentence": "the "`). That way, "The small hall" reads "Explore the small hall" and a relic reads "Find the golden key". If you leave it out, names are always written as they are.

Appendix: every name the build knows

This appendix is written by the build (`python3 build.py --docs`) from the names in the code, so it always matches the engine. Please don't change it by hand: your change would be replaced the next time.

The sections above list most of these already. This appendix puts all of them in one place. The build checks every content file against them and if one of them ever changes, the build's error message lists the current ones.

Conditions (for `found_when` and `unlocked_by`). Counted: `stars`, `three_star_stops`, `stops_gilded`, `stops_won`, `hard_wins`, `trials_finished`, `river_events`, `relics_found`, `suns_forged`, `best_cascade`, `thick_stones_cracked`, `gold_earned`, `gold_held`, `lapis_held`, `win_streak`, `journeys`, `seals_stamped`, `omens_braved` and `chambers_explored`. About the stop just won: `win_moves_to_spare`, `win_on_difficulty`, `win_at_stop`, `win_without_boons`, `win_on_board`, `win_two_specials_at_once`, `win_after_failures`, `win_with_omens`, `win_suns_forged`, `win_best_cascade` and `win_specials_made`. And `relic`, so one relic can unlock the next.

Trial goals: `clear_amulets`, `make_specials`, `cascade`, `gild_in_one_move`, `clear_in_one_move`, `moves_to_spare`, `make_suns`, `make_bands`, `crack_thick`, `gild_half_quickly`, `no_boons` and `combine_specials`.

Badge powers (`effect` in badges): `gild_stones`, `gild_around`, `gild_wide`, `gild_row_and_column`, `row_and_column`, `clear_around`, `clear_its_kind`, `break_covers`, `extra_moves`, `give_lapis`, `lose_moves`, `ungild_stones`, `thicken_stones` and `cover_amulets`.

Hardships (`effect` in curses and omens): `fewer_moves`, `fewer_moves_percent`, `thick_stones`, `thick_stones_share`, `thick_edges`, `fewer_badges`, `more_cursed_badges`, `buried_amulets`, `covered_edges` and `no_boons`.

Boon effects: `extra_moves`, `gild_stones`, `thin_thick_stones`, `cord_row`, `shatter_one`, `shatter_square`, `make_banded`, `make_sun`, `make_ringed` and `shuffle`.

Boons (in rewards, stall items and events): the ids of the files in `content/boons/`, or `"random"` for one drawn at random from those with `"random_reward": true`.

Cover sounds: `sand`, `splash`, `crack`, `stone`, `gild`, `blessing`, `create` and `land`. **Cover rules:** `broken_by` is `beside` or `on`, `layers` goes from 1 to 9, `matches` and `spreads` are `true` or `false`.

Amulets: every picture in `images/amulets/` is an amulet a stop can use (except `sun`, which is a special).

Special amulets' pictures (in `images/specials/`): `band`, `band-glow`, `band-arrows`, `ring`, `ring-glow`, `star-glow` and `star-points`.

Relic icons: any file in `images/icons/relics/` (without its ending), or a picture `images/relics/<id>.png`.

Music: the instruments are `harp`, `lyre`, `oud`, `flute` and `bell`. The scales are `hijaz`, `double harmonic`, `dorian`, `minor`, `major`, `mixolydian`, `lydian`, `phrygian`, `major pentatonic` and `minor pentatonic`, or a list of semitone steps of your own, like `[0, 2, 3, 5, 7, 9, 10]`.

Sounds (to replace with a recording in `sounds/`): `bad`, `blessing`, `bomb`, `coins`, `crack`, `create`, `gems`, `gild`, `land`, `line`, `lose`, `match`, `moves`, `page`, `refund`, `rollup`, `sand`, `select`, `shuffle`, `splash`, `stone`, `sun`, `swap`, `ui`, `unroll` and `win`.

Treasury effects: `extra_moves`, `badge_chance`, `trial_chance`, `win_gold` and `win_lapis`.

Colour changes (in amulet sets): `hue_shift`, `saturation`, `brightness`, `contrast`, `sepia` and `greyscale`, applied in the order you write them (`hue_shift` is in degrees).

Currencies: `gold` and `lapis`.

Kinds of board (for `win_on_board`): `classic`, `grand`, `ruins` and `omega`.

Difficulties (for `win_on_difficulty`): the four `names` in `settings.jsonc` (`difficulty`), which are `Relaxed`, `Normal`, `Hard` and `Hardest` unless your game names its own.

Event goals (`goal.type`): `gild`, `collect` and `score`.

Floor-plan characters: `.` for a gap, `0` for already gilded, `1` for bare stone, `2` for thick stone and each cover's two `letters`.

The starting looks: those named in `edition.jsonc` (`starting_looks`), or otherwise the first file of each kind. Every save begins with them and relies on them, so keep their ids.

Appendix B: Building the desktop and Android apps

This guide covers everything you need to install on a fresh **CachyOS** (Arch-based) machine to build the desktop (Electron) and Android (APK) wrappers, in addition to the plain HTML game.

If you only want to *play* the game, you need nothing but a web browser. Open `dist/amulets-of-anubis.html` and you are done.

What the build produces

Target	Output	Required tools
HTML	`dist/amulets-of-anubis.html`	Python 3
Desktop	`dist/desktop/amulets-of-anubis-<plat>.zip`	Node.js (npx), zip
Android	`dist/android/amulets-of-anubis.apk`	JDK 17 or newer, Android SDK (build-tools 35.0.0 + platform android-35), zip

`./build.sh` runs all targets in sequence and skips any whose tools are missing. You can also target a single step:

```
./build.sh html      # game only
./build.sh desktop   # game + desktop apps
./build.sh android   # game + Android APK
```

Prerequisites (CachyOS / Arch, fish shell)

1. Python 3.9 or newer (for the HTML build)

```
sudo pacman -S python
```

2. Node.js and zip (for the desktop build)

```
sudo pacman -S nodejs zip
```

`electron-packager` downloads the Electron 44.4.3 binaries itself during the build, so you do **not** install Electron system-wide. You just need network access.

3. JDK (for the Android build)

Needed for `javac` (compiles the two small Java source files) and `keytool` (generates the debug signing keystore on first run).

```
sudo pacman -S jdk-openjdk
```

Add the JVM bin directory to your fish PATH (path may vary by version):

```
fish_add_path /usr/lib/jvm/java-26-openjdk/bin
```

4. Android SDK command-line tools (for the Android build)

The `android-sdk` AUR package has a broken PGP signature and should be avoided. Use `android-sdk-cmdline-tools-latest` instead:

```
# AUR helper (if you don't have one)
yay -S yay

# Command-line tools (installs sdkmanager under /opt/android-sdk)
yay -S android-sdk-cmdline-tools-latest
```

This gives you `sdkmanager` at:

```
/opt/android-sdk/cmdline-tools/latest/bin/sdkmanager
```

Add it to your fish PATH:

```
fish_add_path /opt/android-sdk/cmdline-tools/latest/bin
```

5. Install the two SDK components the build calls

The build script invokes four binaries from `build-tools/35.0.0` (`aapt2`, `d8`, `zipalign`, `apksigner`) and reads `android.jar` from `platforms/android-35`. Install both:

```
# /opt/android-sdk is root-owned; make your user own it so sdkmanager can write
sudo chown -R $(whoami) /opt/android-sdk

sdkmanager "build-tools;35.0.0" "platforms;android-35"
yes | sdkmanager --licenses
```

6. Point the build at the SDK

`build.sh` looks for `ANDROID_SDK` (or `ANDROID_HOME` / `ANDROID_SDK_ROOT`) first, then falls back to a list of common paths. Set the variable explicitly:

```
set -gx ANDROID_SDK /opt/android-sdk
echo 'set -gx ANDROID_SDK /opt/android-sdk' >> ~/.config/fish/config.fish
```

Verifying the setup

Run these before building. Each line should print a path (not an error):

```
javac --version
zip --version
ls $ANDROID_SDK/build-tools/35.0.0/aapt2
ls $ANDROID_SDK/platforms/android-35/android.jar
```

Building

```
cd ~/path/to/amulets-of-anubis

# Everything (HTML + desktop + Android + docs + website)
./build.sh

# Or target individual steps
./build.sh desktop
./build.sh android
```

Desktop output

Five zipped apps land in `dist/desktop/`:

File	Platform
`amulets-of-anubis-win32-x64.zip`	Windows, 64-bit
`amulets-of-anubis-linux-x64.zip`	Linux, 64-bit
`amulets-of-anubis-linux-arm64.zip`	Linux on ARM (a Raspberry Pi, for instance)
`amulets-of-anubis-darwin-arm64.zip`	macOS, Apple silicon
`amulets-of-anubis-darwin-x64.zip`	macOS, Intel

Android output

A single APK at `dist/android/amulets-of-anubis.apk`.

Signing: if `platforms/android/release.jks` does not exist, the build auto-generates a debug keystore (`platforms/android/debug.keystore`) and signs with that. The resulting APK is valid and installable. For Play Store distribution or a stable update identity, create a release key first, with its password in a file beside it so the build doesn't

have to ask:

```
python3 -c "import secrets; print(secrets.token_urlsafe(24))" > platforms/android/release.pass
keytool -genkeypair -v \
  -keystore platforms/android/release.jks -storetype PKCS12 \
  -storepass:file platforms/android/release.pass \
  -alias release -keyalg RSA -keysize 4096 -validity 27000 \
  -dname "CN=Amulets of Anubis"
```

The name in `-dname` is written into every APK you sign, where anyone can read it, so use the game's name or a pen name rather than your own. The build picks up `release.jks` and `release.pass` automatically on the next run and `.gitignore` keeps both out of git. **Keep that file safe.** Losing it means you can no longer update existing installs under the same identity.

An unsigned APK (F-Droid)

F-Droid builds the app from this source and signs it with its own key, so it needs the APK unsigned:

```
ANDROID_UNSIGNED=1 ./build.sh android
```

That writes `dist/android/amulets-of-anubis-unsigned.apk` and makes no key. Everything else is the same as the signed build.

The Android build asks nothing and needs no network: Python builds the game, then the SDK's own tools (`aapt2`, `javac`, `d8`, `zipalign`) pack it. It uses these versions and nothing else:

Tool	Version
Android build-tools	35.0.0 (`ANDROID_BUILD_TOOLS`)
Android platform	android-35 (`ANDROID_PLATFORM`); minimum Android 7 (API 24)
JDK	17 or newer (the Java is compiled for Java 8, `--release 8`)
Python	3.9 or newer
zip	any (Info-ZIP)

The same source makes the same APK. Two unsigned builds of one commit are identical, byte for byte, so anyone can check that an APK was built from this source: build it and compare the checksums (`shasum -a 256 dist/android/*.apk`). The build keeps it so by giving the one file it adds to the APK, `classes.dex`, a fixed date.

Version numbers and tags

A release has one version, like `0.9.2`. The source of that release is the tag `v0.9.2`. Android also needs a *version code*, a whole number that must rise with every release, or phones refuse the update. It is made from the version: major × 10000 + minor × 100 + patch, so `0.9.2` is `902`, `0.9.3` is `903` and `1.0.0` is `10000`. (So the minor and patch numbers stay below 100.) Both numbers are written in `platforms/android/AndroidManifest.xml`, where F-Droid looks for them and `build.sh` gives the same ones to `aapt2`.

Each version also has a line for app stores on what changed: `fastlane/metadata/android/en-US/changelogs/<version code>.txt`, in plain words, at most 500 characters. The rest of the store listing (the title, the descriptions, the icon and the phone screenshots) is beside it in `fastlane/metadata/android/en-US/`; `node tools/screenshots/take.mjs store` retakes the screenshots from the built game.

Downloads on GitHub (and Obtainium)

Every release on GitHub has the game and the apps attached, with the version in their names:

File	What
`amulets-of-anubis.html`	the game, one file
`amulets-of-anubis-<version>-android.apk`	the Android app, signed
`amulets-of-anubis-<version>-windows-x64.zip`	Windows
`amulets-of-anubis-<version>-macos-apple-silicon.zip`, `-macos-intel.zip`	macOS
`amulets-of-anubis-<version>-linux-x64.zip`, `-linux-arm64.zip`	Linux

The APK is the only `.apk` in a release, so an app such as Obtainium, given the repository's address, finds and updates it without any settings. The APK attached on GitHub and the one F-Droid makes are signed with different keys, so a phone can't update one with the other: pick one source and keep to it.

A clean build in a container

To check that the Android app builds from a fresh copy with nothing but the tools above, in a throwaway Debian container (Docker or Podman). The first command fetches the tools; the build itself runs with no network. On a Mac with Apple silicon keep `--platform linux/amd64`, because the SDK's Linux tools are only made for x86-64.

```
git clone https://github.com/faiencegames/amulets-of-anubis.git
cd amulets-of-anubis
docker build --platform linux/amd64 -t amulets-android - <<'EOF'
FROM debian:bookworm-slim
RUN apt-get update && apt-get install -y --no-install-recommends \
    openjdk-17-jdk-headless python3 zip unzip curl ca-certificates
RUN curl -fsSLo /tmp/tools.zip \
    https://dl.google.com/android/repository/commandlinetools-linux-11076708_latest.zip \
    && mkdir -p /opt/android-sdk/cmdline-tools \
    && unzip -q /tmp/tools.zip -d /opt/android-sdk/cmdline-tools \
    && mv /opt/android-sdk/cmdline-tools/cmdline-tools /opt/android-sdk/cmdline-tools/latest \
    && yes | /opt/android-sdk/cmdline-tools/latest/bin/sdkmanager --licenses > /dev/null \
    && /opt/android-sdk/cmdline-tools/latest/bin/sdkmanager "build-tools;35.0.0"
"platforms;android-35"
ENV ANDROID_SDK=/opt/android-sdk
EOF
docker run --rm --platform linux/amd64 --network none -v "$PWD":/src -w /src \
    -e ANDROID_UNSIGNED=1 amulets-android ./build.sh android
docker run --rm --platform linux/amd64 --network none -v "$PWD":/src -w /src \
    amulets-android /opt/android-sdk/build-tools/35.0.0/aapt2 dump badging \
    dist/android/amulets-of-anubis-unsigned.apk
```

The last command should begin `package: name='com.amulets.nile'`, with the version code and name of the copy you built.

Troubleshooting

Symptom	Cause	Fix
`[skip] Node.js (npx) or zip not found`	`zip` not installed	`sudo pacman -S zip`
`[skip] Android SDK not found`	`ANDROID_SDK` unset or wrong path	`set -gx ANDROID_SDK /opt/android-sdk`
`[skip] javac not found`	JDK not on PATH	`fish_add_path /usr/lib/jvm/java-26-openjdk/bin`
`AccessDeniedException: /opt/android-sdk/.sdk`	SDK dir is root-owned	`sudo chown -R $(whoami) /opt/android-sdk`
`Skipping non-existent path` from `fish_add_path`	Path doesn't exist yet (e.g. build-tools before sdkmanager runs)	Run `sdkmanager` first, then add the path