



The h3d Script Language

Build scenes in Heimdall3D from a few lines of text

Version 1.0

Heimdall3D

Contents

1	Scope	4
1.1	What a script is	4
1.2	An example	4
2	A first scene	6
3	Notation	9
3.1	Forms in the text	9
3.2	Conformance terms	9
4	Lexical structure	10
4.1	Lines	10
4.2	Words, quoted text and numbers	10
4.3	Names and node references	10
4.4	The version line	11
5	Statements and names	12
5.1	Statements	12
5.2	Setting a name	12
5.3	Using a name	13
6	Points, numbers and anchors	14
6.1	Syntax	14
6.2	Points	14
6.3	Numbers	15
6.4	Seeded numbers	15
6.5	Anchors	15
6.6	When values are worked out	16
7	Loops and waits	17
7.1	Loops over the scene	17
7.2	Loops over numbers	17
7.3	Bodies	18
7.4	Waiting	18
8	Running a script	19
8.1	The check	19
8.2	The run	19
8.3	One undo step	20
8.4	Opening a file	20

9 The verb library	21
9.1 Edit verbs	21
9.2 Script forms	26
10 Reads	28
11 Script files	29
11.1 The file	29
11.2 Running and opening a file	29
11.3 The Scripts card	29
11.4 Any scene as a script	29
11.5 The journal	30
12 Templates	31
A Grammar	32
A.1 Word lists	34
B Errors	35
B.1 The language's refusals	35
B.2 A line's refusals	37
C Conformance	38
D Colours in the notepad	39

1

Scope

This document specifies h3d version 1.0, the script language of Heimdall3D, a 3D viewer. Heimdall3D is a Directory Opus plugin and a standalone viewer, `heimdall3d_view.exe`.

A script builds or changes a scene. It adds models from files, places and turns them, groups and attaches them, adds lights and cameras, sets settings and keys animation. It places models and makes no shapes.

Scripts are written by people and by AI models through the viewer's control pipe, and are read or edited by people.

The language extends the control protocol's edit batch (Chapter 5 gives the one caveat). The language adds names, points, arithmetic, anchors, loops and waits.

It has no `if`, no `while`, no names that change, no string handling and no functions beyond templates. Every script can therefore be checked whole before it runs. The whole language fits one reply of at most 8192 bytes (`list grammar`).

1.1 What a script is

A script is a transaction. The whole text is checked before anything moves. It then runs as one undo step with one reply. A line refused while it runs undoes every line before it. The same script on the same scene gives the same result.

A script is plain text. It is kept in a file with the extension `.h3ds` or sent as one edit request through the pipe.

1.2 An example

The listing below builds a house from a kit of floor, wall and roof models. Every part is placed by the anchors of the parts already placed.

```
h3d 1.0
$floor = add model "kit/floor.glb" as "Floor"
$front = add model "kit/wall.glb" as "Front_wall"
$roof = add model "kit/roof.glb" as "Roof"
place $front at {$floor.top + 0 0 ($floor.size.z/2 - $front.size.z/2)}
$back = duplicate $front as "Back_wall"
place $back at {$floor.top + 0 0 -($floor.size.z/2 - $front.size.z/2)}
$left = duplicate $front as "Left_wall"
turn $left 0 90 0
place $left at {$floor.top + -($floor.size.x/2 - $front.size.z/2) 0 0}
$right = duplicate $left as "Right_wall"
place $right at {$floor.top + ($floor.size.x/2 - $front.size.z/2) 0 0}
place $roof at {$front.top + 0 0 -($floor.size.z/2 - $front.size.z/2)}
```

```
group $floor $front $back $left $right $roof as "House"
```

2

A first scene

This chapter builds a small cottage from a script. The picture below was drawn by Heimdall3D itself from the script you will read. It shows a brick floor, four plaster walls with a door and windows, a tiled roof and a chimney, seen from the street with a sunny sky behind.



The cottage, drawn by Heimdall3D from the script below.

The whole script comes first, so you can see how short it is. After it, five short sections go through it in order. The models come from the free Medieval Village Mega Kit by Quaternius (CC0). Every piece in the kit is built on a 2 metre grid. Floor tiles are 2 by 2 metres, and wall pieces are 2 metres wide and 3.12 metres high. That is why every position in the script is a whole number, and why the roof sits at 3.12.

01-cottage.h3ds

h3d 1.0

A cottage from the village kit: a floor, four walls, corner posts, a roof and a chimney.

The kit is laid out on a two metre grid, so every piece lands on a whole number

.

```

$floor = open "village/Floor_Brick.gltf" replace
add model "village/Floor_Brick.gltf" at 2 0 0
each x in 0..2 step 2:
  add model "village/Floor_Brick.gltf" at {$x 0 2}

# The front: a door and a window, facing the street.
$door = add model "village/Wall_Plaster_Door_Round.gltf" at 0 0 3
$window = add model "village/Wall_Plaster_Window_Wide_Round.gltf" at 2 0 3
# The back and the sides, each turned to face out.
$back = add model "village/Wall_Plaster_Straight.gltf" at 0 0 -1
$back2 = add model "village/Wall_Plaster_Window_Wide_Round.gltf" at 2 0 -1
$west = add model "village/Wall_Plaster_Straight.gltf" at -1 0 0
$west2 = add model "village/Wall_Plaster_Straight.gltf" at -1 0 2
$east = add model "village/Wall_Plaster_Window_Wide_Round.gltf" at 3 0 0
$east2 = add model "village/Wall_Plaster_Straight.gltf" at 3 0 2
wait
turn $back 0 180 0
turn $back2 0 180 0
turn $west 0 -90 0
turn $west2 0 -90 0
turn $east 0 90 0
turn $east2 0 90 0

# A post at each corner, the roof on top and a chimney through it.
each x in -1..3 step 4:
  add model "village/Corner_Exterior_Wood.gltf" at {$x 0 3}
each x in -1..3 step 4:
  add model "village/Corner_Exterior_Wood.gltf" at {$x 0 -1}
$roof = add model "village/Roof_RoundTiles_4x4.gltf" at 1 3.12 1
$chimney = add model "village/Prop_Chimney.gltf" at 2 3.12 0
wait

# A sunny sky and a grass floor, seen from the street a little above.
env = kloofendal_48d_partly_cloudy_puresky
floor.on = 1
floor.material = grass
floor.grid = 0
backdrop.source = 2
camera add Street eye 11 5 13 aim 1 3 1 lens 35
camera look Street

```

The first lines

The first line, `h3d 1.0`, names the version of the language. Lines that start with `#` are comments. They are for you and are skipped when the script runs.

The floor

`open "village/Floor_Brick.gltf" replace` opens the first floor tile as the scene. `add model ... at x y z` then adds another model at the point you give. The `$floor =` at the start of the line keeps what the line made under a name. A name is set once.

The `each x in 0..2 step 2:` line runs the indented line below it for `x` equal to 0 and to 2. The part `{ $x 0 2 }` is a point worked out from the loop's number. A loop body cannot set names. Together with the two tiles added before it, the loop lays a floor of four tiles.

The walls

The front has a door piece and a window piece. The back and the two sides follow. Each wall is added with a name, such as `$back` or `$west`, because later lines turn them. A wall faces out along `+z` as it comes from the kit, so the front needs no turn.

The line `wait` waits until the models being added have finished loading, so the turns that follow can find them. Models are read and decoded off the window's thread, so the window stays responsive meanwhile. Then `turn <node> x y z` turns a wall in degrees about its own axes: 180 for the back, -90 for the west side and 90 for the east side.

The roof and chimney

The roof goes at height 3.12, exactly the height of a wall, so it rests on top. The chimney is added at the same height, a little to one side, and reaches up through the roof. A second `wait` lets both finish loading before the settings lines run.

Sky, floor and camera

The settings lines use the same keys the scene file writes. `env` picks the sky from the list that `list envs` shows. `floor.material` picks the floor surface from `list floors`, and `backdrop.source = 2` shows the sky behind the scene.

`camera add Street eye 11 5 13 aim 1 3 1 lens 35` makes a camera that stands at the eye point, looks at the aim point and has a 35 mm lens. `camera look Street` then looks through it. Heimdall3D checks the whole script before anything moves, and the run is one undo step, so a single undo takes the cottage away again.

3

Notation

The grammar is written in ISO/IEC 14977 EBNF. The sign = defines a rule and , joins parts in order. The sign | separates choices. Square brackets [] mark an optional part, and braces { } repeat a part zero or more times. Round brackets () group. Quoted text is literal, (* *) is a comment, and ? ? describes a character class in words.

Rule names are lower case with underscores. Appendix A holds the whole grammar. The word lists in it are generated from the viewer's own tables. They cover the anchors, axes, queries, leaves and verbs. The grammar is read in two layers, the script and the statement, as Appendix A describes.

3.1 Forms in the text

In the form of a verb, angle brackets name what is written there, as in <node> and <name>. Square brackets mark what may be left out. A bar separates choices. The letters x y z stand for three numbers.

Examples are set in monospace. Each is a whole line as a script writes it.

3.2 Conformance terms

The word *must* states a requirement on a viewer that implements this version. The word *refuses* means that the viewer answers with the error named, in its exact words (Appendix B), and changes nothing.

4

Lexical structure

4.1 Lines

A script is a sequence of lines separated by line feeds. A carriage return before a line feed is ignored. Lines are numbered from 1, and blank and comment lines count. Every error names its line by that number.

A line is trimmed of spaces and tabs at both ends. A blank line is skipped. A comment is a whole line whose first character after the indent is `#`, and it is skipped too. There are no comments at the end of a line. One statement stands on a line.

A line that begins with a space or a tab is indented. Only the body of an `each` is indented (Chapter 7). An indented line anywhere else is refused as the error `indent`. A blank line or a comment line inside a loop's body does not end the body.

4.2 Words, quoted text and numbers

Words are separated by spaces or tabs. A word starts with a letter or `_` and continues with letters, digits and `_`.

In a verb line, an argument is any run of characters other than spaces and tabs, or text in double quotes. Quoted text is one argument and may hold spaces. Inside quoted text, `\` stands for a quotation mark and `\\` for a backslash. Any other backslash is itself. A quote left open is refused with the words "a quote is not closed" (the line refusal `quote-open`, Appendix B).

A number is digits with an optional fraction and an optional exponent, such as `1`, `0.25`, `.5`, `5.` and `2e3`. A sign before a number is a minus applied to it, and a range may start or end on a signed number, as in `-2..2`. Hexadecimal is read nowhere.

A number that does not read inside a point or brackets is refused as the error `not-number`. A number past the reach of a 32-bit float, such as `1e39` or `1e400`, is refused as `not-number` inside a point or brackets. A verb's own number arguments are refused in that verb's words when the line runs. Numbers are kept as 32-bit floats. A value nearer zero than `1e-12` is written as `0`.

4.3 Names and node references

A *name* is `$` followed by a letter or `_`, then letters, digits or `_`, as in `$floor` and `$front_wall`. A malformed name is refused as the error `name-form`. Names inside quoted text are not names.

A node is referred to in one of three ways, in verb arguments as well as in points and brackets:

- by its number, as in `#3`;
- by a name that holds it, as in `$front`;
- by its label, quoted as in `"Front wall"` or written as a word as in `Chair`.

An anchor written after a name, as in `$floor.top`, is worked out outside braces and brackets only when the word after the dot is an anchor. Otherwise the dotted word stays as it is, so `$lamp.intensity` becomes `#7.intensity`. An anchor written after a number or a label, as in `#3.top`, `"Front wall".top` and `Chair.top`, is worked out only inside braces or brackets.

4.4 The version line

The version line stands on the first statement line. Comment and blank lines may come before it. It is the word `h3d`, a space, then the version. More spaces or tabs may follow that first space. Anywhere else it is refused as the error `version-place`.

The version must be exactly `1.0`. Any other is refused as the error `version`. The language version is separate from the control protocol's version.

5

Statements and names

5.1 Statements

A statement is one of the following:

- a scene setting, written `<key> = <value>` as the scene file writes it, such as `tone.gamma = 2.2;`
- a node's leaf, written `obj.#3.t = 0 1 0` or `set <node>.<leaf> = <value>;`
- a verb line, which is an English verb and its arguments (Chapter 9);
- a name set, written `$name = <statement>;`
- `wait.`

In a script, the characters `$`, `{`, and `(` or `-(` that opens a word or follows `=`, are read as values. This holds everywhere outside quoted text, in a setting's or a leaf's value too. A value that holds such a character as text is quoted. Apart from that, every edit line of the control protocol is a statement.

5.2 Setting a name

`$name = <statement>` runs the statement and keeps what it made under the name. That is a node, or a job's result when the statement starts a job. `add model` is a job, and the name holds the node once the model has landed.

The spaces round `=` may be left out. A line that starts with `$` sets a name only when no anchor and no leaf stands before its `=`. A node's leaf is set through a name with `set`, as in `set $lamp.intensity = 2`. A bare `key = value` is a scene setting.

A name is set once per script. The following are refused:

- setting a name again, as the error `set-twice`, which names the line that set it first;
- using a name never set, as the error `unset`;
- a name set with nothing after `=`, as the error `empty-set`;
- a statement that made nothing to name, as the error `made-nothing`, when it runs.

A name with an anchor after it, such as `$floor.top`, is used and never set. Names belong to the script. A template's names and parameters are its own.

5.3 Using a name

Wherever a node is written, `$name` stands for the node it holds. Inside a point or arithmetic it stands for the node's centre, or for an anchor of it, as in `$floor.top`. A loop's name holds each node or number in turn.

A line that uses a name a job is still making waits for that job before it runs.

```
$table = add model "table.glb" as "Table"  
$lamp = add light spot as "Lamp" at 0 3 2 aim $table  
set $lamp.intensity = 2
```

6

Points, numbers and anchors

Two kinds of value are worked out before a line runs: a *point* in braces and a number in brackets. Both may use numbers, names, node references and anchors with +, -, *, / and brackets.

6.1 Syntax

```
point      = "{" , [ gap ] , group , { gap , ( "+" | "-" ) , gap , group } , [
    gap ] , "}" ;
group      = item | item , gap , item , gap , item ;
item       = item_factor , { ( "+" | "-" | "*" | "/" ) , item_factor } ;
item_factor = "-" , item_factor | arithmetic | call | number | reference ;
arithmetic = "(" , [ gap ] , expression , [ gap ] , ")" ;
expression = term , { [ gap ] , ( "+" | "-" ) , [ gap ] , term } ;
term       = factor , { [ gap ] , ( "*" | "/" ) , [ gap ] , factor } ;
factor     = "-" , [ gap ] , factor | arithmetic | call | number | reference ;
call       = "random" , [ gap ] , "(" , [ gap ] , expression , [ gap ] , "," , [
    gap ] , expression , [ gap ] ,
    "," , [ gap ] , expression , [ gap ] , ")"
    | "ground" , [ gap ] , "(" , [ gap ] , reference , [ gap ] , "," , [
    gap ] , expression , [ gap ] ,
    "," , [ gap ] , expression , [ gap ] , ")" ;
reference  = ( "$" , name | "#" , name_chars | raw_quoted | name ) , [ "." ,
    anchor , [ "." , axis ] ] ;
```

6.2 Points

Braces hold one or more groups joined by + or - with spaces round them. Inside braces, spaces separate a group's items, so an item holds no space outside its brackets or its quoted label. "Front wall".top is one item. A group is one item or three items. The groups are added and taken away from the left. A group sign stands between two groups. A sign at the end of a point, or two signs together, is refused as the error point-form.

An item is a number, a node reference with an optional anchor and part, or brackets. An item may also be these joined by +, -, * and / with no spaces between them, with a minus before any of them. The value of a one-item group is a point. The three items of a three-item group are numbers.

{0 1 0}, {\$table.top + 0 0.1 0} and {\$a.centre - \$b.centre} are points. {(#4.top + #5.top)/2} is one item. {\$a.top * 2} is three items and is refused.

A point that is not of this form is refused as the error point-form. A point added to a number is refused as the error mixed.

6.3 Numbers

Round brackets hold an expression whose value is a number. It is made of numbers, names holding numbers, an anchor's part such as `.size.z` or `.top.y`, the signs `+`, `-`, `*` and `/`, and brackets. Multiplication and division come before addition. A unary minus is allowed. An example is `turn $door 0 ($i * 15) 0`.

The value of brackets may be a point. It is then written as three numbers.

A division by zero is refused as the error `divide-zero`. A bracket left open is refused as the error `unclosed`. An expression that is empty or has something after its end is refused as the error `number-form`.

6.4 Seeded numbers

The call `random(low, high, seed)` may stand wherever a number may, inside round brackets or inside a point's braces. Its value is a number from `low` up to but not including `high`. Both `low` and `high` are numbers or expressions whose value is a number.

The same three arguments give the same number on every run and on every machine. The number comes from the seed alone, by the SplitMix64 mix of the seed's 64-bit floating-point value.

The seed may be a number, such as a loop's name holding a number, so each pass of a loop draws its own number. The seed may also be a point, such as a node's name, so each node draws its own number from where it stands. An example is `turn $tree 0 (random(0, 360, $i)) 0`.

A call with other than three arguments is refused as the error `number-form`. A `low` or `high` given as a point is refused as the error `mixed`.

6.5 Anchors

An *anchor* is a point of a node's box in the world, as the node stands when the line runs. The box is the axis-aligned box round the node's corners as it is placed in the world. The anchors are the ones Box snapping meets. A node written without an anchor stands for its centre.

The parts `.x`, `.y` and `.z` after an anchor take one part of it as a number. At most one part stands after an anchor. A name with an anchor, written outside braces and brackets, takes every further dotted word into its anchor, so `$x.top.w` is refused as the error `axis`.

`top` and `bottom` follow the scene's up axis, `Y` or `Z`. With `Y` up, `front` is the `+Z` face and `back` the `-Z` face. With `Z` up, `front` is the `-Y` face and `back` the `+Y` face. `left` is the face at the smallest `X` and `right` the face at the largest `X`.

An anchor that is not one of the language's is refused as the error `anchor` inside braces and brackets. Outside them, `$a.tip` stays as text (`#7.tip`) and its verb refuses it when the line runs. A part that is not `x`, `y` or `z` is refused as the error `axis`. An anchor on a name that holds a number is refused as the error `not-node`. A node with no box, because nothing is drawn, is refused as the error `no-box` when the line runs.

Anchor	Meaning
<code>.top .bottom</code>	the centres of the upper and lower faces of the node's box, along the scene's up axis
<code>.centre</code>	the middle of the box
<code>.front .back</code>	the centres of the front and back faces, as the paragraph above gives them for each up axis
<code>.left .right</code>	the centres of the faces at the smallest and largest <code>X</code>
<code>.pivot</code>	the node's placement origin in the world
<code>.size</code>	the width, height and depth of the box, as a point

6.6 When values are worked out

Every value is worked out against the scene as the lines before it left it, just before its own line runs. A value that needs a model that a job is still adding waits for the job.

At the check, before anything runs, every line is worked out with stand-ins, and every name stands for a node. The check refuses every fault of form:

- an unclosed brace, bracket or quote;
- a malformed name;
- a stray character;
- a number that does not read;
- a point whose groups do not hold one or three items;
- a bracket that is empty or has something after its end;
- an anchor inside braces or brackets, or a part, that is not one of the language's.

Faults that depend on values are refused when the line runs:

- a point added to a number (mixed);
- a division by zero;
- an anchor on a name that holds a number;
- a one-item group whose value is a number;
- a node with no box;
- a node that does not exist.

7

Loops and waits

7.1 Loops over the scene

`each <name> in <query> [where name ~ <text> | where name = <text>]` runs the indented lines below once for each node, light, group, camera or bone the query finds. The name holds that item.

The queries are:

- `objects`, which finds every node: models, groups, pieces and lights;
- `models`, which finds the models;
- `lights`, which finds the lights placed in the scene;
- `groups`, which finds the groups;
- `cameras`, which finds the scene's cameras;
- `bones`, which finds the bones.

A query that is not one of these is refused as the error `each-query`. A head that is not of this form is refused as the error `each-form`.

`where name =` matches the whole name. `where name ~` without a `*` matches a name that holds the text anywhere. With a `*` the pattern covers the whole name, and `*` stands for any run of characters. `lamp*` matches "Lamp 2" and does not match "My lamp". Both `~` and `=` ignore case. The name matched is the node's own label when it has one, and otherwise its row's name.

A query loop is put out against the scene when it is reached, after the script's jobs so far have ended.

```
each l in lights where name ~ "lamp*":  
  set $l.intensity = 2
```

7.2 Loops over numbers

`each <name> in <a>..<b> [step <s>]` runs the indented lines for `a`, `a+s`, `a+2s` and on, while the number has not passed `b`. The step `s` is 1, or -1 when `b` is below `a`. The loop reaches `b` only when `b` falls on a step, and `a..a` runs once. A range splits at its first `...`. Three dots together are refused as the error `each-form`.

A step of zero is refused as the error `step-zero`. A step whose sign points away from `b` is refused as the error `step-way`. Number loops are unrolled at the check.

```
each i in 0..4:  
  turn #3 0 ($i * 15) 0
```

7.3 Bodies

The *body* is the indented lines under the head. It needs at least one line, or it is refused as the error `empty-each`. Bodies do not nest, and a nested body is refused as the error `nested`. The loop's name exists only inside its body.

A body may not set a name, because it would set the name each time round. Such a line is refused as the error `each-sets`. A `wait` in a body waits each time round.

Unrolled lines count against the cap of 4096 lines that a script may unroll to. The check counts range loops in full and each query loop's body once. While the script runs, the lines a query loop or a template puts out count as they are put out. A script over the cap is refused as the error `cap`.

7.4 Waiting

`wait` waits until every job the script has started has ended. Lines that use a job's result wait for that job on their own. `wait` serves a line that needs all of the jobs, such as a key after a motion.

8

Running a script

8.1 The check

Before anything moves, the whole script is checked. The check refuses exactly the following:

- a wrong version line;
- a version line that is not the first statement (`version-place`);
- an indented line outside a loop (`indent`);
- a name that is not set once and before it is used;
- a loop head whose form is wrong (`each-form`);
- a query that is not one of the language's (`each-query`);
- a loop with no body (`empty-each`);
- a loop in a loop's body (`nested`);
- a body line that sets a name (`each-sets`);
- a step of zero (`step-zero`);
- a step pointing away from the range's end (`step-way`);
- a line whose form is wrong, with its names stood in for, which covers an unknown verb, a malformed setting or leaf, an unclosed quote, brace or bracket, and the faults of form that Chapter 6 lists;
- a script over the cap.

Faults that depend on the scene are refused when their line runs.

Every fault is answered at once, one line each, in the script's order and in the error table's words. Nothing is applied. A dry run checks and answers checked N steps; nothing was applied.

8.2 The run

A script that passes runs as a job of its own, a line at a time. Each line's values are worked out against the scene the lines before it made.

A line that starts a job, such as adding a model, a motion or a rig fit, lets the next lines run unless they use its result. One reply comes when the script ends, and it names how many lines the script applied.

While a script runs, another edit is answered busy. Reads, looks and job requests go through.

8.3 One undo step

The whole script is one undo step. One undo takes back everything it did.

A line refused while the script runs, a job that fails, or a cancel of the script's job undoes every line applied before it. This happens once the script's own jobs have ended. The reply names the line, as in line 4: no node is called "Nobody".

8.4 Opening a file

The first statement of a script is the first open reached before any line has applied. `open <path>` starts from that file without the arrangement and keys it was last left with. After a model opens, the settings go to their defaults. When the file is a Gaussian splat capture that opens as a place (one with a spawn, its author's or one found automatically), the camera and the floor and grid are the exception: the camera stands at the capture's spawn with the capture lens, and the floor and grid stay off, as when a person opens the same file. A scene file (`.h3scene`) brings its own settings. The same script therefore makes the same scene on any viewer.

When the scene holds work, `open <path> replace` is needed. A script that does not start with open needs a model open. With none it is refused as the error `no-model`.

`undo`, `redo` and `open` anywhere else in a script are refused as the error `replaces` when their line runs, because the script could not undo them. A `template` line of any form (`save`, `list`, `show` or `drop`) is refused as the error `template-save` when it runs. A file opened in the window while a script runs ends the script, which is refused as the error `replaced`.

9

The verb library

These are the verbs a script line may start with, except that a script refuses `undo`, `redo` and every `template` line when it runs, and `open` except as its first statement. Each has its form, what it does and one example. The list is generated from the viewer's own verb table. A verb's arguments are given by its form. The viewer checks them when the line runs and refuses, in the verb's own words, arguments that do not fit.

Besides verbs, a line may set a scene key or a node's leaf. A scene key is written `<key> = <value>`, as the scene file writes it. A leaf is written `obj.#<id>.<leaf> = <value>` or `set <node>.<leaf> = <value>`.

A node's leaves are:

- `t`, with `x y z`;
- `r`, a quaternion `x y z w`;
- `s`, with `x y z`;
- `base`, sixteen numbers;
- `name`;
- `hidden`, 0 or 1;
- `id`;
- `light`;
- for a placed light, intensity, colour (`r g b`) and range.

The keys `env`, `floor.material`, `floor.user`, `variant` and `display` take a name from `list envs`, `list floors`, `list variants` or `list modes`.

A card's control is named by the manual's page, then the section when its words repeat, then its words. The words are quoted when they hold a space.

9.1 Edit verbs

undo

steps the scene back one undo step; stands in a batch of its own.

Example: `undo`

redo

steps the scene forward one undo step; stands in a batch of its own.

Example: `redo`

remove <node>

takes a node, and what hangs under it, out of the scene.

Example: remove #4

duplicate <node> [as <name>]

a copy of the node and what hangs under it, beside it, sharing its file.

Example: duplicate \$front as "Back wall"

group <node> <node> ... [as <name>]

a group node over the nodes, where the first stood.

Example: group \$floor \$front \$roof as "House"

ungroup <node>

the group's members back under its parent, where they stand; the group goes.

Example: ungroup "House"

attach <node> to <node> [joint <name>]

the first follows the second, or a rigged figure's joint, from where it stands.

Example: attach "Lamp" to "Desk"

detach <node>

a node that follows another lets go where it stands.

Example: detach "Lamp"

place <node> at {point}

puts the node's bottom centre at the point.

Example: place \$front at {\$floor.top + 0 0 4}

turn <node> x y z

turns the node by degrees about its own axes, X then Y then Z, where it stands.

Example: turn \$left 0 90 0

on floor <node>

sets the node down on the floor.

Example: on floor \$chair

lay flat

turns the opened file onto its largest flat face and sets it on the floor.

Example: lay flat

select <node> | none

picks the node as a press on its row does, or nothing.

Example: select "Chair"

add model <path> [as <name>] [at x y z]

adds a model as a job, read and decoded off the window's thread; add model lines stand in a batch of their own.

Example: \$floor = add model "kit/floor.glb" as "Floor"

add light point|spot|sun|area [as <name>] [at x y z] [aim x y z | aim <node>]

a placed light, lighting what it aims at as the unit key does; over the model when no place is given.

Example: add light spot as Key at 2 3 2 aim \$subject

**add terrain [as <name>] [at x y z] [size w d] [height h] [seed n] [rough 0-1]
[cells n] [shape fbm|ridged|billow] [octaves n] [lacunarity f] [warp f]
[features f] [erode n] [inertia f] [capacity f] [dissolve f] [settle f] [dry
f] [weather n] [talus deg] [slide f] [colour r g b] [bare | surface <floor|**

#rrgbb> [heights lo hi] [slopes lo hi] [soft h s] [breakup b] [tile m] [strength s] [flow f] [deposit d] ... [image <path>]

ground from seeded noise, or from a height image's brightness, the same on every run, carved by n steps of water and settled by n of sliding; under the model when no place is given; a noise terrain is dressed in up to 8 surfaces (floor materials or #rrgbb colours) shaped by height, slope and erosion, or left 'bare'.

Example: \$ground = add terrain size 40 40 height 3 seed 7 erode 200

scatter <node> [count n] [over <terrain> | size w d] [seed n] [turn a b] [scale a b] [as <name>] [at x y z]

copies of the node's model laid out by the seed, standing on the terrain's ground or across an area, the same on every run.

Example: \$trees = scatter \$tree count 300 over \$ground seed 4 scale 0.8 1.3

aim <light> at x y z | aim <light> at <node>

turns a placed light to face a point or a node's centre.

Example: aim "Key" at "Chair"

promote [rig|file] <name> [<index>]

makes a rig or file light a placed light where it stands, its own switched off.

Example: promote rig "Key"

light [rig|file|glowing] <name> [<index>] on|off | ... shadow on|off

a light's own switch or its shadow, as its row in the list sets them; list lights names them.

Example: light rig "Fill" off

key <property> [on <layer>] <seconds> <value> [<shape>] [ease in|out|inout] [tangent <mode>] [in dt dv] [out dt dv] [breakdown|hold|extreme]; ...

keys on the property's track, in the animation file's own fields: <node> tx ty tz rx ry rz sx sy sz vis; a light node's intensity red green blue cone range size; light.<n>.<on azimuth elevation distance intensity size red green blue>; bone <name> rx ry rz (degrees over what plays) tx ty tz; camera <name> tx ty tz ax ay az lens; control <page>.<words>.

Example: key "Chair" ty 0 0; key "Chair" ty 2 1

unkey <property> [<seconds>]

takes the key at that time off the property, or all of them.

Example: unkey "Chair" ty 2

time <seconds>

puts the clock there, as a scrub does.

Example: time 1.5

play

plays the timeline.

Example: play

stop

stops it where it is.

Example: stop

range <start frame> <end frame> [rate <fps>] [loop once|loop|pingpong]

the timeline's range, rate and end.

Example: range 0 120 rate 30 loop

layer add <name> [additive] | layer <name> on|off|remove | layer <name> weight <w>

the animation's layers: what a key goes in, whether it plays, and how much of it.

Example: layer add Bounce additive

drive light.<n> by <node> [near d] [far d] [at-near v] [at-far v] | drive light.<n> off

the light's intensity from its distance to the node, as the Lighting card's Drive by sets it.

Example: drive light.0 by "Chair" near 1 far 5

camera add <name> eye x y z aim x y z [follow <node>]|<node> [lens mm] [parallel] [unkeyed]

a scene camera, keyed where it stands and cut to, as the Cameras card's Add makes one from the view.

Example: camera add Hero eye 3 2 3 aim \$chair lens 50

camera set <name> [eye x y z] [aim x y z|<node>] [lens mm] [parallel|perspective] [at <seconds>]

the camera's pose keyed at that time, the clock's when none is given.

Example: camera set Hero eye 4 2 3 at 2

camera look <name> | camera look on | camera look off

looks through that camera whatever the cut row says, through the cut row's cameras, or at the pane's own view.

Example: camera look Hero

camera cut <name> at <seconds>

the cut row names that camera from then on, and the pane follows the cut row.

Example: camera cut Hero at 0

camera aim <name> at x y z|<node>

points the camera at a point, or follows a node.

Example: camera aim Hero at "Chair"

camera drop <name>

takes the camera and its keys away.

Example: camera drop Hero

frame <node>

turns and brings in the pane's own camera until the node fills the frame.

Example: frame \$house

autorig [shape] [guides <Name> x y z, ...] [landmarks x y z; ...]

fits a skeleton to the opened model as a job, as the Rig card's Fit does; guides put the joints a person would drag (Head, Neck, Hips, ShoulderL/R, ElbowL/R, WristL/R, KneeL/R, AnkleL/R); alone in its batch.

Example: autorig

joint <bone> at x y z [alone] | joint <bone> centre|refit|mirror

moves a joint of the skeleton, its children with it unless alone; the skin re-binds as a job.

Example: joint Head at 0 1.7 0

bone add <bone> [as <name>] [at x y z] | bone remove <bone>

a bone hung under another, or taken out.

Example: bone add Tail as Tail2 at 0 0.9 -0.4

motion <node> <library name | path> [clip <n>] [as <label>]

a clip driven onto that character alone, as a job, played on its own clock beside the others.

Example: motion \$hero "Walk"

actor <node> [clip <n>] [start <seconds>] [speed <x>] [loop on|off]

a character's own clock: which of its clips, when it starts, how fast, and whether it loops or holds its end.

Example: actor \$hero start 1 speed 1.2

rig reach|falloff <bone> <amount> | rig rigid on|off | rig remove | rig character

the skin's reach per bone and its rigid binding (a re-bind job), the fit taken off, or the model kept as a character.

Example: rig reach Spine 0.8

open <path> [replace]

opens a model or a scene file as a job, alone in its batch; replace is needed when the scene holds work.

Example: open "scenes/room.h3scene" replace

still <stem> [<width>x<height>] [samples <n>]

a still from the pane's camera as a job, written as <stem>.png in the export folder; the Capture card's size and samples when none are given; paid.

Example: still front 1920x1080

screenshot <stem> [card <page>] [png]

the pane as a person sees it, strip, cards and docks included, as a job: <stem>.jpg (or .png) in the export folder; card <page> opens that page's card for the picture and keeps the card alone; free, up to 1024 pixels across without a licence.

Example: screenshot lighting card lighting

record <stem> turntable|clip [png|apng|gif|mp4] [pane|512|720p|1080p] [fps <n>] [frames <n>]

a recording as a job into the export folder, one file or a folder of frames for png; what is given is set on the card first; paid.

Example: record spin turntable mp4 720p

sprite <stem>

a sprite sheet and its sidecar as a job into the export folder, as the Sprites card is set; paid.

Example: sprite walker

save scene <stem>

the scene file as a job, <stem>.h3scene in the export folder.

Example: save scene house

destination <file>

the shot template that file holds becomes the Shots card's destination as it stands, unsaved, as a job; stands in a batch of its own.

Example: destination "fiverr edited.ini"

journal start [<stem>] | journal stop

each step taken from then on, written as the script lines that take the scene there, <stem>.h3ds in the export folder.

Example: journal start session

script save <stem>

the scene as the script that makes it again, <stem>.h3ds in the export folder, as a job.

Example: script save house

script run <file> | script open <file>

a script file run as a job, or opened in the Scripts card's notepad; a relative path stands beside the open file.

Example: script run "house.h3ds"

export model <stem> [glb|fbx]

the model, its rig and its clips as a job into the export folder; paid.

Example: export model house glb

paint begin [bone <name>] | paint bone <name> | paint brush [draw|add|subtract|smooth|fill] [weight w] [strength s] [smoothing m] [radius <px>] [surface on|off] [mirror on|off] | paint at x y z; ... [radius r] | paint look <id> x y; ... | paint fill x y z | paint fill look <id> x y | paint lock|unlock <bone> | paint undo|redo | paint end keep|draft|discard

weight painting as the paint rail does it, with the panel's brush: a stroke line is one stroke of the painter's history, its dabs at points on the model at rest or at a look's pixels; paint lines stand in a batch of their own.

Example: paint at 0 1 0 radius 0.1

template save "<name>" (<param>, <param>=<default>, ...) <lines> end | template list | template show "<name>" | template drop "<name>"

a stored workflow: its lines as an Edit carries them, \$param where an argument goes and {<node> + x y z} for a place beside a node; one file each in Documents\Heimdall3D\Templates beside the shipped ones, which a file of the same name replaces; save alone in its batch, ended by end; list names each with its parameters.

Example: template save "Shelf" (\$width=1) ... end

run "<name>" [<argument> ...] [<param>=<value> ...]

a template's lines with the arguments put in, as one batch: validated whole, one undo step; templates run templates four deep, never themselves.

Example: run "Three-point lighting" \$chair reach=3

set <page>.[<section>].<words> = <value>

a card control by the manual's page, section and words: a slider's or a stepper's number, a switch's 0 or 1, a row's cell by its words, a colour's linear r g b.

Example: set lighting.Key.Softness = 0.4

press <page>.[<section>].<words>

a switch, a cell or a button on a card, as a person presses it; <page>.Reset puts the card back.

Example: press rendering.Reset

9.2 Script forms

h3d <version>

the language version the script is written in; a viewer refuses one it does not speak.

Example: h3d 1.0

\$<name> = <statement>

keeps what the statement made, a node or a job, under a name; set once.

Example: \$floor = add model "kit/floor.glb" as "Floor"

\$<name>[.<anchor>[.x|.y|.z]]

a name in use; a node's anchors: top bottom centre front back left right pivot size.

Example: turn \$left 0 90 0

{<point>}

a point: three numbers, or points and anchors added and taken away, worked out before the line runs.

Example: place \$roof at {\$front.top + 0 0 -2}

(<arithmetic>)

a number worked out from numbers, names and anchors with + - * / and brackets.

Example: turn \$door 0 (\$i * 15) 0

random(<low>, <high>, <seed>)

a number from low up to high, the same for the same seed on every run; a loop's number or a node's place as the seed makes each differ.

Example: turn \$tree 0 (random(0, 360, \$i)) 0

ground(<terrain>, <x>, <z>)

how high the terrain's ground stands at x and z across the level axes, so a model placed there stands on it.

Example: place \$tree at {\$x (ground(\$g, \$x, \$z)) \$z}

each <name> in <query> [where name ~|= <text>]:

the indented lines below once for each node, light, camera or bone the query finds; ~ holds, * any run.

Example: each l in lights where name ~ "lamp*":

each <name> in <a>..** [step <s>]:**

the indented lines below once for each number from a to b.

Example: each i in 0..4:

wait

waits for every job the script has started.

Example: wait

10

Reads

A *read* is asked apart from a script, one line to a request, and changes nothing. Its answer is lines of text. `list grammar` answers the whole language in one reply.

get <pattern> [<pattern> ...] | get keys <target>

the scene's lines whose keys match; * matches any run, `obj.#7.*` one node's; keys gives a target's tracks as the animation file writes them.

Example: `get obj.#7.*`

list objects | boxes | keys | animation | bones | cameras | clips | lights |

layers | drivers | motions | actors | envs | floors | variants | modes |

controls [<page>] | digests | paint | weights x y z | grammar

one short line per item; controls alone names the pages; envs, floors, variants and modes give the names `env`, `floor.material`, `floor.user`, `variant` and `display` take; digests a word for the scene keys, each node and the keys, which moves when they do; boxes each node's box in the world, as a script's anchors read it; keys every track as its key line; animation the animation file's own lines, which the keys' word hashes.

Example: `list objects`

changed

the lines that differ from this connection's last read.

Example: `changed`

status

revision, what is busy, jobs, counts, the camera and the clock.

Example: `status`

11

Script files

11.1 The file

A script file is plain UTF-8 text with the extension `.h3ds`, in any case. It holds at most 4 MiB (4,194,304 bytes). A byte-order mark at its start is ignored.

A relative path in `add model` and in a first-line `open` stands beside the script's own file. A shipped script has no file of its own, so its relative paths stand beside the file the viewer has open. A relative path given to `script run` or `script open` through the pipe stands beside the file the viewer has open. `heimdall3d_view.exe --call script run|open <file>` makes a relative path whole from the shell's working folder.

11.2 Running and opening a file

`script run <file>` reads the file on a worker and runs it as a job. `script open <file>` shows it in the Scripts card's notepad and runs nothing. `script save <stem>` writes the scene as the script that makes it again. The file is `<stem>.h3ds` in the export folder, or `<stem>_1.h3ds`, `<stem>_2.h3ds` and on when that name is taken. It is never written over a file that is there.

`heimdall3d_view.exe --call script run|open <file>` does the same from a shell and waits for the run. `--call script` writes `scene.h3ds` into the export folder the same way and prints the script.

```
heimdall3d_view.exe --call script run "C:\Scripts\house.h3ds"
```

11.3 The Scripts card

In the viewer, the Scripts card lists the scripts in `Documents\Heimdall3D\Scripts`, or in the folder `HEIMDALL_SCRIPT_DIR` names. It lists the ones that ship with Heimdall3D and the ones opened from elsewhere. Each is a tile with a picture of its last run. A press runs one. A script opened while the viewer shows nothing opens the demo scene first.

The card's notepad edits, runs and saves a script. A `.h3ds` file dropped on the viewer, or opened from Directory Opus or Explorer, opens in the notepad ready to run.

11.4 Any scene as a script

Any scene the viewer holds can be written as a script that makes it again. The script holds, in the scene's order, its first file opened and each model, light, group and attachment made. It records where each stands: turn in degrees where that is exact, and otherwise the node's own `t`, `r`, `s` and base lines.

It also holds layers, cameras, the settings that differ from their defaults, keys, the range, motions and drives.

What no line reaches is written as a comment. Run on an empty viewer, the script gives back the same scene.

11.5 The journal

`journal start [<stem>]` writes the scene as a script. After each step a person or a model takes, it then writes the lines that take the scene to the next step, under `# step N`. `journal stop` ends it.

`heimdall3d_view.exe --journal <file>` journals a window's steps into a file. Replayed on an empty viewer, a journal reaches the same scene.

12

Templates

A *template* is a named, stored list of lines with parameters. It is saved with `template save "<name>" (<param>, <param>=<default>, ...) <lines> end.` `template list`, `template show "<name>"` and `template drop "<name>"` list, show and remove templates.

`run "<name>" [<argument> ...] [<param>=<value> ...]` puts the template's lines in its place, with each `$param` filled in. A point given to a template is one argument, as in `run "Lift" Chair to={0 2 0}`.

Templates run at most 4 deep, and deeper is refused as the error `template-deep`. A template that runs itself is refused when it is saved, in the words "the template `<name>` runs itself". A script runs templates and refuses every template line (`save`, `list`, `show` and `drop`) when the line runs, as the error `template-save`.

The template's own names are kept apart from the script's. A refusal inside a template names it, as in inside the template `<name>: <why>`.

Templates ship with Heimdall3D: three-point lighting, a product shot and a turntable. A user's template of the same name replaces a shipped one.

```
each i in 1..2:
  run "Lift" Right to={0 ($i * 2) 0}
```

A

Grammar

This appendix holds the whole grammar, then the word lists generated from the viewer's tables. The grammar is read in two layers, as the viewer reads a script. The rule `script` gives a script's lines, its loops, its names and where values stand in a line. Each statement line, once its values are worked out, is read by the rule `statement`. A read is the rule `read_line`.

A line's values are worked out first. A name that holds a node becomes that node's number (#7). A point in braces becomes its three numbers, separated by spaces. Brackets become their number, or three numbers when their value is a point. `$name.anchor` becomes three numbers, or one with a part. In a run line a point stays one argument. It becomes the three numbers in quotes, and `name={...}` becomes `"name=x y z"`.

The grammar gives the form. The chapters give the rules the form cannot. The version is exactly 1.0. A name is set once and before it is used. A loop's body sets no name. A step is not zero and points towards the end of its range. A script keeps within the cap of 4096 lines. At most one part, x, y or z, follows an anchor. A node's number is at most 4294967295. A number stays within a 32-bit float. A name's anchor written outside braces takes every dotted word after it, and at most one may follow the anchor, a part.

```
(* The h3d script language in ISO 14977 EBNF, the exception "-" included. The word
   lists (anchor, axis, query,
   leaf, the key words and the verbs) are generated from the viewer's tables into
   generated/words.ebnf. A script
   is read in two layers, as the viewer reads it: the script layer finds its lines,
   loops, names and the values in
   them; once a line's values are worked out to plain numbers and nodes, the statement
   layer reads what is left.
   A test holds both layers to the parser with strings each must take and strings each
   must refuse. *)
```

```
(* Script lines *)
script      = { skip_line } , [ version_line ] , { skip_line | statement_line |
               each_block } ;
skip_line   = [ gap ] , [ "#" , { text_char } ] , end_of_line ;
version_line = [ gap ] , "h3d_" , [ gap ] , version , [ gap ] , end_of_line ;
version     = digits , "." , digits ;
statement_line = script_statement , [ gap ] , end_of_line ;
end_of_line = newline | ?_the_end_of_the_text_? ;

(* Script statements *)
script_statement = "wait" | name_set | value_line - ( ( "each_" | "h3d_" ) , {
               text_char } | name_set_like ) ;
(* A line that starts with $ and has an = with no anchor or leaf before it sets a name
   , or is refused. *)
name_set_like   = "$" , { text_char - ( "." | "=" ) } , "=" , { text_char } ;
name_set        = "$" , [ gap ] , name , [ gap ] , "=" , [ gap ] , value_line ;
```

```

value_line      = line_word , { gap , line_word } ;
(* A bracket opens arithmetic at the start of a word and after "="; elsewhere it is a
   character of the word. *)
line_word       = ( arithmetic | "-" , arithmetic ) , { word_part } , word_end
                 | ( word_part , { word_part } , word_end ) - ( ( "(" | "-(" ) , {
                   text_char } )
                 | "=" , word_end ;
word_end        = { "=" } , [ "-" ] ;
word_part       = raw_quoted | point | name_value | eq_part | plain_run | paren_run ;
eq_part         = "=" , ( arithmetic | "-" , arithmetic | "-" , ( word_part - ( "(" ,
                   { text_char } ) ) )
                 | word_part - ( ( "(" | "-" ) , { text_char } ) ) ;
plain_run       = plain_char , { plain_char | "(" } ;
paren_run       = "(" , { plain_char | "(" } ;
plain_char      = text_char - ( gap_char | "'" | "{" | "$" | "=" | "(" ) ;
raw_quoted      = "'" , { raw_quoted_char | "\" , ( ( "'" | "\" | raw_quoted_char ) ) ,
                   "'" ;
(* A name's anchor takes every dotted word after it; at most one, a part, is allowed (
   chapter 5). *)
name_value      = "$" , ( name , [ "." , anchor , { " , name_chars } ] ) ;

(* Loops *)
each_block      = each_head , { skip_line , body_line , { body_line , skip_line } ;
each_head       = "each " , [ gap , name , gap , "in" , gap , ( query_source , range
                   ) , [ gap , " : " , gap ] ,
                   end_of_line ;
query_source    = query , [ gap , "where" , gap , "name" , gap , ( "~" | "=" ) , gap ,
                   ( raw_quoted | bare ) ] ;
range          = ( signed_number , ".." , signed_number ) - ( { text_char } , "..." ,
                   { text_char } ) ;
[ gap , "step" , gap , signed_number ] ;
body_line      = gap , ( "wait" | value_line - ( ( "each " | "h3d " ) , { text_char }
                   | name_set_like ) ) , [ gap ] ,
                   end_of_line ;

(* Values *)
point          = " ( " , [ gap ] , group , { gap , ( "+" | "-" ) , gap , group } , [ gap
                   gap ] , " )" ;
group          = item | item , gap , item , gap , item ;
item           = item_factor , { ( "+" | "-" | "*" | "/" ) , item_factor } ;
item_factor    = "-" , item_factor | arithmetic | call | number | reference ;
arithmetic     = "(" , [ gap ] , expression , [ gap ] , " )" ;
expression     = term , { [ gap ] , ( "+" | "-" ) , [ gap ] , term } ;
term           = factor , { [ gap ] , ( "*" | "/" ) , [ gap ] , factor } ;
factor         = "-" , [ gap ] , factor | arithmetic | call | number | reference ;
call           = "random" , [ gap ] , " ( " , [ gap ] , expression , [ gap ] , " , " , [ gap
                   gap ] , expression , [ gap ] ,
                   " , " , [ gap ] , expression , [ gap ] , " )" ;
[ gap ] , "ground" , [ gap ] , " ( " , [ gap ] , reference , [ gap ] , " , " , [ gap
                   gap ] , expression , [ gap ] ,
                   " , " , [ gap ] , expression , [ gap ] , " )" ;
reference      = ( "$" , name | "#" , name_chars | raw_quoted | name ) , [ gap ] ,
                   anchor , [ gap ] , " , " , axis ] ;

(* Statements *)
statement      = control_set | control_press | node_set | setting | verb_line ;
verb_line      = ( edit_verb - ( "set" | "press" ) ) , { gap , word } ;
read_line      = read_verb , { gap , word } ;

```

```

word=====quoted_,_{quoted_}_,{bare_}_bare_;
control_set===== "set"_,_gap_,_(node_path_|control_path_)_,_gap_,_"="_,_
    value_text_;
control_press===== "press"_,_gap_,_control_path_,_gap_;
control_path===== (segment_-(_"#"_,_{text_char_}_)"obj."_,_{text_char_}_)__,_"."
    _,_path_part_,_[_"."_,_path_part_]_;
path_part=====quoted_|segment_;
segment=====[_gap_]_,_segment_char_,_{segment_char_|gap_char_}_;
segment_char=====text_char_-(gap_char_|_"_|'"'_|_"=")_;
node_path=====[_]"obj."_]_,_(node_id_|quoted_)_,_"."_,_leaf_;
node_set=====[_]"f."_]_,_"obj."_,_(node_id_|quoted_)_,_"."_,_leaf_,_gap_,_
    ="_,_value_text_;
node_id=====_"#"_,_{_"0"_]_,_(digit_-"0"_)_,_{digit_}_;
setting=====setting_key_,_gap_,_"="_,_value_text_;
setting_key===== (key_char_,_{key_char_|gap_char_}_)
=====-(_"set"_|_"press"_)_,_([gap_char_|_"=")__,_{text_char_}_]_|_[
    f."_]_,_"obj."_,_{text_char_}_
=====|_"f."_,_gap_char_]_)
=====|_[_"f."_]_,_"obj.count"_;
key_char=====text_char_-(gap_char_|_"_|'"'_|_"=")_;
value_text=====_{text_char_}_;

(*_Words_*)
name===== (letter_|_"_"_)_,_{letter_|digit_|_"_"_}_;
name_chars===== (letter_|digit_|_"_"_)_,_{letter_|digit_|_"_"_}_;
quoted===== "'"_,_{quoted_char_|"\",_(("'"_|_"\")_|quoted_char_)_,_"'"_;
bare===== (text_char_-(gap_char_|_"_|'"'_|_"=")__,_{text_char_|gap_char_}_;
number===== (digits_,_[_"."_,_[digits_]_]_|_[_"."_,_digits_]_,_[exponent_]_;
signed_number===== [_]"+"_|_"-"_]_,_number_;
exponent===== (_"e"_|_"E"_)_,_[_"+"_|_"-"_]_,_digits_;
digits=====digit_,_{digit_}_;
gap=====gap_char_,_{gap_char_}_;
gap_char===== " "_|_?a_|horizontal_|tab_|_;
letter===== ?a_|letter_|A_|to_|Z_|or_|a_|to_|z_|_;
digit===== ?a_|digit_|0_|to_|9_|_;
newline===== ?a_|line_|feed_,_after_,_an_|optional_|carriage_|return_|_;
text_char===== ?any_|character_|but_|a_|line_|feed_|_;
raw_quoted_char===== ?any_|character_|but_|a_|quotation_|mark_,_a_|backslash_|or_|a_|line_|feed_|_;
quoted_char===== ?any_|character_|but_|a_|quotation_|mark_,_a_|backslash_|or_|a_|line_|feed_|_;

```

A.1 Word lists

```

(* Generated from the parser's tables by test_control_language.cpp; do not edit. *)
anchor = "centre" | "top" | "bottom" | "front" | "back" | "left" | "right" | "pivot" |
    "size" ;
axis = "x" | "y" | "z" ;
query = "objects" | "models" | "lights" | "groups" | "cameras" | "bones" ;
leaf = "t" | "r" | "s" | "name" | "hidden" | "light" | "intensity" | "colour" | "range"
    | "base" | "id" ;
edit_verb = "undo" | "redo" | "remove" | "duplicate" | "group" | "ungroup" | "attach"
    | "detach" | "place" | "turn" | "on" | "lay" | "select" | "add" | "scatter" | "aim"
    | "promote" | "light" | "key" | "unkey" | "time" | "play" | "stop" | "range" | "
    layer" | "drive" | "camera" | "frame" | "autorig" | "joint" | "bone" | "motion" |
    "actor" | "rig" | "open" | "still" | "screenshot" | "record" | "sprite" | "save" |
    "destination" | "journal" | "script" | "export" | "paint" | "template" | "run" |
    "set" | "press" ;
read_verb = "get" | "list" | "changed" | "status" ;

```

B

Errors

A refusal that belongs to a line is written `line N: <words>`. A refusal that belongs to the script as a whole carries no line number. These are `no-model`, `busy`, `replaced`, `cancelled`, `caller-gone`, `made-nothing` when the name waits for a job and the job lands with nothing to name, and a job that fails after its line has run. A `job-failed` whose job fails at its own line carries that line.

The marks `{1}`, `{2}` and `{3}` in the words stand for the name, line, value or word concerned. The viewer fills them in.

The two tables below are generated from the viewer's own tables. The first lists the script language's refusals. The second lists the refusals a line itself answers, which are its words, a node it names and a control it names.

Every verb also refuses, in its own words and when the line runs, arguments that do not fit its form, a node or file that is not there, and a scene it cannot act on. Those refusals are not listed here.

A quote left open on a line is refused with the words "a quote is not closed" (`quote-open`). A quote left open inside a point or brackets is refused with the `unclosed words`.

B.1 The language's refusals

Code	Words
<code>version</code>	this viewer speaks <code>h3d {1}</code> , not <code>h3d {2}</code>
<code>version-place</code>	<code>h3d {1}</code> opens a script, before its first line
<code>unset</code>	<code>#{1}</code> was never set
<code>set-twice</code>	<code>#{1}</code> is set already, on line <code>{2}</code>
<code>indent</code>	an indented line belongs to an each
<code>nested</code>	an each's lines hold no each
<code>empty-each</code>	an each needs indented lines under it
<code>each-form</code>	each <code><name></code> in <code><query></code> [where <code>name ~ = <text></code>]: or each <code><name></code> in <code><a>..<code></code></code> [<code>step <s></code>]:
<code>each-query</code>	each goes over objects, models, lights, groups, cameras or bones
<code>each-sets</code>	a name is set once; an each would set <code>#{1}</code> each time round
<code>step-zero</code>	a step of zero never ends

Code	Words
step-way	a step of {1} never reaches {2}
empty-set	\${1} = <statement>
stray	"{1}" has no meaning here
not-number	{1} is not a number
cap	the script unrolls past {1} lines
name-form	a name is \$ and a letter, then letters, digits or _
point-form	{ } is a point: three numbers, or points added and taken away
number-form	() is a number: {1}
not-node	\${1} is a number and has no .{2}
anchor	{1} has no anchor .{2}; a node's anchors are top bottom centre front back left right pivot size
axis	.{1} is a point; its parts are .x .y and .z
divide-zero	a division by zero
mixed	a point and a number do not add
unclosed	a {1} opened is not closed
made-nothing	\${1}: the line made nothing to name
job-failed	job {1}: {2}
template-save	a script runs templates; it saves none
template-deep	templates run more than {1} deep
cancelled	cancelled
caller-gone	its caller went away
empty-line	the line is empty once its names are put in
run-form	run "<name>" [<argument> ...] [<param>=<value> ...]
no-template	no template is called {1}; template list names them
inside	inside the template {1}: {2}
busy	a script is running; job wait on it, or cancel it
checked	checked {1} steps; nothing was applied
no-model	no model is open
not-a-node	{1} is not a node
no-box	{1} has no box to place by

Code	Words
missed	{1}
replaces	{1} replaces the scene, which a script cannot undo; it stands in a batch of its own
replaced	the scene was replaced while the script ran

B.2 A line's refusals

Code	Words
quote-open	a quote is not closed
no-verb	no verb is called {1}
no-key	an assignment names no key
set-form	set <page>.<words> = <value>
press-value	press takes no value
control-path	a control is named <page>.<words> or <page>.<section>.<words>
node-form	a node is named obj.#id.<leaf> or obj."name".<leaf>
no-leaf	a node has no {1}; its leaves are {2}
no-node-id	no node is {1}
no-node-name	no node is called "{1}"
nodes-named	{1} nodes are called "{2}":{3}
no-page	no page is called {1}; list controls names them
no-control	no control on {1} is called {2}{3}



Conformance

A viewer conforms to h3d 1.0 when it does all of the following:

- It accepts every script that has the grammar's form, keeps the rules the chapters give, and whose verb lines fit their verbs' forms. It refuses every other script, with the words of Appendix B or the verb's own.
- It checks a script whole before applying any of it.
- It refuses every fault of form at the check, before anything moves.
- It applies a script as one undo step and undoes what it applied when a later line is refused.
- It works out every value against the scene as the earlier lines left it.
- It names, by its line, every refusal that belongs to a line.
- It answers `list grammar` within 8192 bytes.

The reference implementation is Heimdall3D's viewer. Its tests check every example in this document.

D

Colours in the notepad

The Scripts card's notepad colours a script as it is typed. Each word takes the colour of what the h3d language makes of it, so a verb added to the language is coloured as a verb with no change to the notepad.

What it is	Colour	Example
Verb: what a line does	 E8A33D	add terrain
Word: the language's own smaller words	 9CA0A8	as at size height
Number, or a colour written as # and six or eight hex digits with at least one letter among them	 A8D48F	40 -20 2.5 1..5 #8a6a4a
Text in double quotes	 D9A6E8	"village/Floor_Brick.glTF"
Comment: a line starting with #	 7D828B	# A valley
Name: a \$ name, a node by number (# and digits only), anything after a dot. A line whose first character is # is a comment, so a node by number is written after a verb, as in place #1 at 0 0 0	 76A8E6	\$valley #3 .top
Setting: a key set with =, a control path after set or press	 6FC3C0	floor.on obj.#3.t
Function: a word followed by an opening bracket with no space; height (36) is a word followed by a worked-out number	 E8C86D	ground(
Anything else	 ECEEF2	Valley

A word that starts with a digit, such as 1080p, is not a number. In \$valley = add terrain ... both add and terrain are verbs, because a statement after \$name = starts a new line's verb.

The example script from the first scene, as the notepad shows it:

```
h3d 1.0
# A valley: ground from seeded noise, carved by water.
$floor = open "village/Floor_Brick.gltf" replace
$valley = add terrain as Valley at 0 0 0 size 240 240 height 36 seed 11
wait
place $floor at {6 (ground("Valley", 6, 10)) 10}
env = kloofendal_48d_partly_cloudy_puresky
floor.on = 0
camera add Valley eye 70 70 170 aim -20 12 30 lens 40
camera look Valley
```

A long line stays one line: the text scrolls sideways to keep the cursor in view, and lines do not wrap. A script of a hundred thousand lines stays quick to type in and scroll, since only the lines in view are drawn and coloured, and a single very long line stays quick too.

Index

actor, 24
add, 22
aim, 23
anchor, 15
anchor, 36
attach, 22
autorig, 24
axis, 36

body, 18
bone, 24
busy, 36

caller-gone, 36
camera, 24
cancelled, 36
cap, 36
changed, 28
checked, 36
control-path, 37

destination, 25
detach, 22
divide-zero, 36
drive, 23
duplicate, 22

each-form, 35
each-query, 35
each-sets, 35
empty-each, 35
empty-line, 36
empty-set, 36
export, 25

frame, 24

get, 28
group, 22

indent, 35
inside, 36

job-failed, 36
joint, 24
journal, 25

key, 23

lay, 22
layer, 23
light, 23
list, 28

made-nothing, 36
missed, 37
mixed, 36
motion, 24
must, 9

name, 10
name-form, 36
nested, 35
no-box, 36
no-control, 37
no-key, 37
no-leaf, 37
no-model, 36
no-node-id, 37
no-node-name, 37
no-page, 37
no-template, 36
no-verb, 37
node-form, 37
nodes-named, 37
not-a-node, 36
not-node, 36
not-number, 36
number-form, 36

on, 22
open, 25

paint, 25
place, 22
play, 23
point, 14
point-form, 36
press, 26
press-value, 37
promote, 23

quote-open, 37

range, 23

read, 28

record, 25

redo, 21

refuses, 9

remove, 21

replaced, 37

replaces, 37

rig, 24

run, 26

run-form, 36

save, 25

scatter, 23

screenshot, 25

script, 25

select, 22

set, 26

set-form, 37

set-twice, 35

sprite, 25

status, 28

step-way, 36

step-zero, 35

still, 25

stop, 23

stray, 36

template, 31

template, 26

template-deep, 36

template-save, 36

time, 23

turn, 22

unclosed, 36

undo, 21

ungroup, 22

unkey, 23

unset, 35

version, 35

version-place, 35