in silico Plants, 2025, 7, diaf005
https://doi.org/10.1093/insilicoplants/diaf005
Advance access publication 28 April 2025
Technical Advance

in silico Plants
OXFORD

Technical Advance
Functional–Structural Plant Models

The Virtual Plant Laboratory: a modern plant modelling framework in Julia

Alejandro Morales[1,*,‡], David B. Kottelenberg[1,‡], Ana Ernst[1], Rémi Vezy[2], Jochem B. Evers[1]

[1]Centre for Crop Systems Analysis, Wageningen University & Research, Bornsesteeg 48, Building 109, 6708 PE Wageningen, The Netherlands
[2]CIRAD, UMR AMAP, *TA A51/PS2* 34398 Montpellier cedex 5, France

[*]Corresponding author. Centre for Crop Systems Analysis, Wageningen University & Research, Bornsesteeg 48, Building 109, 6708 PE Wageningen, The Netherlands. E-mail: alejandro.moralessierra@wur.nl.
[‡]These authors should be considered joint first authors.

Handling Editor: Amy Marshall-Colon

Abstract The Virtual Plant Laboratory (VPL) is a software for building, simulating, and visualizing functional–structural plant (FSP) models. FSP models focus on the interactions between plant structure, internal physiological processes, and the biotic and abiotic environment. VPL is built in the Julia programming language and is designed to be a flexible and extensible platform for FSP modelling. Using Julia brings the advantage that only one programming language is required for the whole modelling cycle as Julia is as fast as compiled languages but also dynamic as interpreted languages. VPL provides a graph rewriting system for building dynamic models of plant growth and development, an interactive 3D visualization system and a Monte Carlo ray tracer for simulating light interception by plant canopies. In this paper, we introduce VPL, highlighting the main components, modelling paradigms, and design decisions behind it, as well as a future roadmap for further development. We also present a model for intercropping of legumes and cereals that was built fully with VPL, as an example of what can be built with this software. VPL is open source and available in all common operating systems for anyone to use. Full documentation and tutorials are available at https://virtualplantlab.com.

KEYWORDS: functional–structural plant modelling; ray tracing; intercropping; plant architecture; computational ecophysiology.

1. INTRODUCTION

The Virtual Plant Laboratory (VPL) is a software for building, simulating, and visualizing functional–structural plant (FSP) models. FSP modelling focuses on the interactions between plant structure, internal physiological processes, and the biotic and abiotic environment (Godin and Sinoquet 2005; Buck-Sorlin 2013). FSP models span across multiple biological scales, from cells up to plant communities (Louarn and Song 2020), and temporal scales, from calculations of instantaneous photosynthetic rates (Pearcy et al. 2005; Wang et al. 2017) to the growth and development throughout a plant's life cycle (de Reffye et al. 2021). This makes FSP modelling an indispensable tool for (i) scaling up plant physiological processes, (ii) understanding the interplay between plant structure, physiology, and biotic and abiotic environment, or (iii) addressing questions in ecology or plant production where the 3D structure and topology of plants play a significant role, in contrast to classic process-based plant models in which plant structure is not described explicitly (Vos et al. 2009).

In addition to covering multiple scales and addressing 3D plant structure explicitly, FSP models may also be interpreted as a special case of agent-based modelling (Zhang and DeAngelis 2020). At each scale, FSP models can describe the different relevant structures (e.g. cells, organs, branches, plants) as separate entities, with an internal state and behaviour. These entities interact with each other directly, through topological connections between organs or cells, or indirectly, through the environment. The dynamic interplay between direct and indirect interactions of these different entities results in emerging properties such as competition between plants for resources, which make FSP models a useful tool to explain complex spatial-temporal patterns in plant systems from simpler, local patterns.

These features that characterize FSP models also make them more challenging to implement than traditional mathematical or simulation models. For example, computing the interception of light by an explicit 3D structure requires using algorithms from computational geometry, many of which were originally developed for computer graphics, such as ray tracing (Bailey 2018) or radiosity (Chelle and Andrieu 1998). Similarly, keeping track of the topological relationships among organs of a plant over time requires using data structures, such as graphs (Godin and Caraglio 1998; Hemmerling et al. 2008), whereas physiology and environmental physics are generally modelled by systems of algebraic and/or (partial) differential equations (Farquhar et al. 1980; Baldocchi et al. 2002). In contrast, a traditional process-based model of plant growth is often only comprised of a system of algebraic equations iterated over time (Wang et al. 2002; van Ittersum et al. 2003).

Given this need for multiple, non-trivial data structures and algorithms, specialized scientific software solutions (hereafter denoted as *modelling frameworks*) have been developed for FSP modelling, including GroIMP (Hemmerling et al. 2008), OpenAlea (Pradal et al. 2008), and L-Studio (Prusinkiewicz et al. 2000) among others (see below for details). These modelling frameworks allow modellers to focus on translating their scientific knowledge into code and execute it, without having to learn all the mathematical and computational details mentioned above. Modelling frameworks are fundamental to the recent growth in FSP modelling. Further progress in the field is, to some extent, limited by the improvement and maintenance of said frameworks. The VPL (https://virtualplantlab.com) is a novel FSP modelling framework built in the Julia programming language aimed at allowing easier maintenance and further improvement.

In its current iteration, VPL focuses on FSP models defined at the levels of organ, plant, and canopy (Buck-Sorlin 2013). VPL is not intended to be used for models that describe the anatomy of organs (i.e. where individual cells or tissues are represented explicitly), such as existing 3D anatomical models of roots (Postma et al. 2017; Couvreur et al. 2018), leaves (Ho et al. 2016; Retta et al. 2023), or meristems (Smith et al. 2006). Similarly, VPL does not currently allow representing canopies with voxels or crown envelope volumes, which are often used when simulating trees (Sinoquet et al. 2001; Chen et al. 2008; Duursma and Medlyn 2012).

In this paper, we first introduce the VPL (version 0.0.6, released in September 2024), highlighting the main components, modelling paradigms, and design decisions behind it, as well as a future roadmap for further development. We also briefly describe the implementation of VPL in the Julia programming language. Finally, we present a short case study of a model for intercropping of legumes and cereals that was built with VPL, as an example of what can be built with this software. VPL is fully open source (all code is hosted in the VirtualPlantLab GitHub organization: https://github.com/VirtualPlantLab) and available in all common computing platforms for anyone to use (Linux, Windows, MacOS, and FreeBSD). We refer potential users to the official website https://virtualplantlab.com for instructions on installation and use.

2. THE VIRTUAL PLANT LABORATORY

2.1 Modelling paradigms and features

2.1.1 Plant topology (graphs).

VPL uses directed acyclic graphs, also known as tree graphs, to describe the topological structure of a plant. Canonically, each node in a graph represents either an organ or a special topological feature that separates organs, such as the stem node that separates internodes, although this is not enforced. For instance, nodes can also be used to specify the origin and orientation of a plant or sub-plant scales, such as branches or tillers. A node in a graph may contain any data the user deems relevant, encapsulated within user-defined data types. Typically, these data will describe the geometry of an organ and its internal state like age, physiological signals, absorbed light, etc. It is also possible to store data at the graph level, which is intended to represent variables and parameters that are constant across organs or that are defined at the plant level, such as plant biomass. Constants and common parameters may also be defined as global variables outside of graphs.

In a dynamic FSP model, the graphs will evolve over time, either by changing the data stored in the nodes or by adding, subtracting, and/or replacing nodes in the graph. To facilitate this dynamic evolution, VPL offers *rules* and *queries*. In both cases, a subset of the nodes in the graph will be selected based on a matching condition. This condition is defined by the type of node to be selected and a user-defined condition function that will be executed on each node of the right type, and which should return true or false. The condition function will have access to the data stored in the node, in the graph as well as all the other nodes in the graph through their topological relationships. The latter allows selecting nodes based on their context within the graph.

In the case of rules, the selected nodes will be replaced by a new node, a sub-graph (i.e. a set of connected nodes) or nothing, in which case the node and all the nodes downstream of it will be removed. The new node or sub-graph is created by a second user-defined function that has access to the same information as the condition function. This process of replacement of nodes is generally known as *graph rewriting*, such as the one used in the software GroIMP (Kurth et al. 2005) and multiple rules may be applied to a graph in an iterative fashion. These rules are conceptually defined to be executed in parallel, which offers the advantage of efficient processing. The only cost is that within an iteration, two rules cannot match the same node and all node matching must occur before any node is replaced to ensure safe parallelism. As a result, rules cannot be ordered, and must be applied collectively.

In the case of queries, the selected nodes are returned in an array that the user can use to modify the internal state of those nodes. Queries do not rewrite the graph and they can be applied in any order and at any point within a simulation. Rules are used when changes to the topology of the plant are needed (e.g. emergence of new organs, pruning, senescence, etc.), whereas queries are used when only the internal state of existing nodes need to be adjusted (e.g. updating the dimensions of organs due to growth).

A graph in VPL is thus defined by three components: (i) an *axiom*, which is the initial set of nodes and their topological connections, (ii) a series of rules (if the model is dynamic) and, optionally, (iii) a user-defined object to store graph-level data. Queries are defined outside of graphs and are then applied to existing graphs to retrieve arrays of matched nodes.

VPL does not make any assumptions regarding time steps in the simulation. For example, it is possible to trigger the graph rewriting rules multiple times during a simulated day if the user wishes to do so. Generally, modifying the graphs over time thus consists of applications of rules, application of queries, and modification of the internal states of the nodes matched by queries. Graphs may also be modified by the ray tracer (see below) which will affect the internal state of nodes in order to store absorbed power.

The graph rewriting paradigm described in the above can be considered to lie between the Lindenmayer systems of string rewriting or L-systems (Prusinkiewicz and Lindenmayer 1990) and relational growth grammars as used in GroIMP (RGG, Hemmerling et al. 2008) However, a major departure from these paradigms is the fact that VPL does not use a domain-specific language (DSL) to define rules and queries but rather a procedural programming paradigm based on user-defined functions, data types, and methods to traverse the graph and modify internal states, which relies on normal Julia syntax. The main difference is that a DSL specifies what the computer program should achieve without indicating how to do it (in the background, an algorithm will generate the code to be executed). In a procedural programming approach, the user gives the computer the specific instructions to be followed.

We do not have evidence of which system is superior, nor a clear set of criteria to make such evaluation. Our experience is that a procedural programming approach is more transparent and facilitates debugging as the user writes the code that will be executed, as opposed to the modelling platform generating the code in the background.

To further clarify the different philosophies, let's imagine a graph that consists of two types of nodes (denoted types F and A). We would like to identify all the nodes of type F that have a parent node of type A. In the XL language used within GroIMP (Kurth et al. 2005), the user would use a pattern such as:

(* A --> *) F

Where the '`-->`' denotes any type of edge between two nodes, and the '`(*...*)`' indicates that the first part of the pattern is being used as context (i.e. we are actually interested in the F node either to be extracted in a query or to be replaced in a rule). In context-sensitive L-systems (Prusinkiewicz and Lindenmayer 1990), such as the one used in the language L+C, part of L-Studio (Prusinkiewicz et al. 2000), one could use:

A < F

Where '<' denotes that A should precede F on the string to be rewritten, but only F is being replaced. For the same tasks, in VPL, one would define a function such as:

```
function match_node(node)
  # Extract the parent of the current
tested node
  p = parent(node)
  # Check if the data stored in the
node is of type A or not
  if data(p) == A then
  return true
   else
  return false
   end
  end
```

This function could then be used in the creation of a `Rule` or `Query` object that targets nodes of type F. VPL provides functions to test different relationships (`parent`, `children`, `ancestor`, `descendant`, see VPL documentation for details) and built-in Julia tools for iteration (e.g. for loops) and conditional checking (e.g. if statements) can then be used to build complex patterns.

An exception to this procedural programming approach is the possibility in VPL to build sub-graphs for the axiom or the rules of a graph, using a simple algebra for graph construction. This algebra defines two operators ('+' and brackets '()') and greatly simplifies the construction of graphs while enforcing the constraints of no cycles in the graph and every node only having one parent (except the root node). See Supplementary Equation S2.1 for an example of a rule using the algebra for graph construction.

Finally, although VPL graphs do not encode hierarchical scales directly, nodes may represent subsets of the plant and hence graphs in VPL are compatible with the Multiscale Tree Graph formalism (MTG, Godin and Caraglio 1998), the main difference being that the user would be responsible of keeping track of the different scales manually. Full compatibility with MTG, including the commonly used file format for data exchange, has not been incorporated yet but will be included in future releases of VPL.

2.1.2 Plant structure (geometry).

VPL can convert a graph into a 3D representation, i.e. a series of triangular meshes, by using procedural geometry generation using the so-called turtle graphics approach (Verhoeff 2010), although the 3D structure is not only used for visualization. The *turtle* traverses the graph from the root node and execute a user-defined method to generate the geometry associated to each node in the graph. These methods are responsible for creating 3D meshes representing the structure of the plants, as well as colour objects (for visualization, see below) and spectral *materials* (for the ray tracer, see below). These methods can also affect the orientation and position of the turtle, i.e. its state, which is then kept for when it visits the next node in the graph. For simplicity, VPL offers multiple methods to modify the state of the turtle as well as generate the 3D meshes corresponding to common geometry primitives.

The turtle approach allows defining geometry locally, such as the angle of a leaf being specified relative to the branch it connects to rather than the horizontal plane, and the final geometry being generated will depend on the topological connections among nodes and the changes in the state of the turtle. This allows modelling complex plant architecture as an emergent property of simpler local geometry.

The generated 3D meshes alongside colour and material objects are stored in a scene object which can then be used for ray tracing or visualization. In addition to generating these scenes from graphs, it is also possible to generate those 3D meshes directly and add them to an existing scene, alongside corresponding colour and material objects. This is particularly useful when adding a soil layer to be able to simulate soil reflectance, or more complex environments such as the structure of a greenhouse compartment (Buck-Sorlin et al. 2011). VPL also allows exporting these 3D meshes into common file formats (e.g. PLY or OBJ) as well as importing them and we have successfully used exported 3D meshes for 3D printing. Special methods are offered by VPL to scale and rotate imported meshes so that they can be used more easily in the context of a model.

One additional feature that has been included in VPL is the ability to generate different 3D meshes and/or colour and material objects within the same simulation. Possible cases include: (i) different degree of geometric detail for visualization and ray tracing, (ii) removing some of the structure for specific visualizations, (iii) different ray tracers and therefore material objects, or (iv) visualizing the same scene with different colour schemes that map to properties stored in the nodes of the graphs. The latter allows 3D visualization to be used for the purpose of displaying organ properties like nitrogen content or absorbed light.

2.1.3 Light interception (ray tracing).

In general, VPL only focuses on the algorithms and data structures that are specific to topology and structure in FSP modelling rather than on the functionality of said models, which the user can implement using a range of packages in Julia. However, the calculation of light interception by plants at the level of individual organs or below is sufficiently complex that VPL comes with a solution for it. Several algorithms have been used in the past to model light interception by 3D plants, including ray tracing (Cieslak et al. 2008; Wang et al. 2017; Bailey 2018), radiosity (Chelle and Andrieu 1998), and rasterization (Vezy et al. 2020). In its current iteration, VPL offers a ray tracing algorithm.

The ray tracer within VPL allows for multiple wavelengths or wavebands to be simulated simultaneously, using the method proposed by Cieslak et al. (2008). VPL does not make any assumptions about which wavelengths are being simulated, nor the physical units in which light is expressed. Multiple light sources are available (point, line, area, and directional) and several spectral materials are offered (black, Lambertian, Phong, and sensors that register light but do not affect it). New materials and light sources can be user-defined without modifying the source code of VPL, ensuring versatility.

The ray tracer may run in parallel across multiple cores, it is stochastic (different pseudo-random number generators may be used) and employs the Russian roulette mechanism proposed by Cieslak et al. (2008) to avoid errors in the energy balance when terminating rays. The intersection between rays and triangles is accelerated by creating a bounding volume hierarchy on the triangular meshes using surface area heuristic (MacDonald and Booth 1990). This ensures that a minimum number of triangles need to be tested against each ray, reducing the overall computational cost.

Often, FSP models are approximating a canopy, whether growing in the field or under controlled conditions. In such cases, simulating hundreds of individual plant is often not feasible, so a smaller plot is typically modelled, although such downsizing can result in boundary effects when calculating light interception. To avoid this, the ray tracer of VPL implements a *grid cloner* based on the notion of graphics instancing (Marschner and Shirley 2018), similar to the modelling framework GroIMP. This is an approach that emulates copying the scene a number of times along the different coordinate axes (horizontally or vertically). However, to avoid excessive memory consumption and computation from actually copying the meshes, graphics instancing will instead modify the origin of the ray being traced as if that ray was entering a neighbouring clone of the scene. Note that the location of the virtual clones does not necessarily match the bounds of the scene, but rather it would be related to the spacing between plants. This means that the situation in which neighbouring plants overlap is captured correctly by graphic instancing as opposed to, for example, periodic boundaries.

2.1.4 Visualization (rendering).

VPL offers the possibility to render graphs (in 2D) and the 3D scenes generated from graphs and/or manually. For 3D scenes, it is also possible to render some of the elements used in the ray tracer, such as bounding boxes and light sources. Scenes can be rendered using a variety of backends from the Makie Julia package. In fact, VPL will return a data structure created by Makie which can be further modified by the user, so all the features in said package will be available for visualizations in VPL.

As of the time of writing, these backends include OpenGL, WebGL, and Cairo. The latter can only generate static vector graphics so it is not useful for exploration of 3D scenes but it will work on headless systems that have no support for 3D graphics, and generally produce highest-quality plots for publications. The OpenGL and WebGL backends allow interacting with 3D scenes (rotation, panning, and zooming) and 2D scenes (panning and zooming). WebGL is useful when running VPL in a web context such as Jupyter notebooks whereas OpenGL is the preferred option for local simulations. For OpenGL and WebGL backends, it is possible to export snapshots of the scene and create animations with scripted cameras.

Graph visualization will use a default node label (based on the type and internal node id) and colour. However, the user can edit these properties for each type of node. In the case of 3D scenes, it is possible to visualize the wireframe pattern of the meshes (i.e. the edges of all the triangles) as well as the normal vectors for each triangle.

2.2 Future roadmap

VPL is in constant development as new features are added to address the needs of the FSP modelling community. We do have the following roadmap which determines the current priorities for further development of VPL:

- Compatibility with MTG. There is currently a Julia implementation of MTG [the MultiScaleTreeGraph.jl package (Vezy 2023a)] which will serve as a starting point. Our aim is to be able to merge our current paradigm for graphs with the multiscale nature of MTG by adding new types of edges between nodes and adjusting the graph construction algebra. We expect this will make VPL more useful for simulations of larger, more complex plant structures (e.g. mature trees), especially when the simulation is initialized at an advanced stage of development. It may also be advantageous to have an explicit mechanism for storing data at different scales within the plant.
- Additional algorithms for light interception based on previous efforts. This is intended to reduce the computational

cost of simulating the distribution of light within canopies in scenarios where the degree of realism that ray tracing ensures is not required. We note that the usual shortcuts that are used to reduce the computational burden of ray tracing (e.g. reduce the number of rays, coarse meshing of plant structure, coarse integration over the day, etc.) introduce numerical errors in the calculations of daily canopy photosynthesis [see Kent and Bailey (2021) for an example] in which cases faster, simpler approaches may be justified.

- Mapping graphs to systems of ordinary differential equations (ODEs) for the purposes of simulating internal transport, such as for plant hydraulics, assimilate flow, hormonal signals, etc. Julia already offered state-of-the-art support for ODE systems, and there already examples of deriving ODE models from graphs, such as the package NetworkDynamics.jl (Lindner et al. 2021).
- Interface between the 3D structure of plants with grids to simulate plant–environment interactions beyond light (e.g. temperature profiles within a canopy, soil water and heat transport, etc.). This would be achieved by *slicing* the scene along one or more axes and identifying the structural components in each of the resulting layers or voxels, both above and belowground. This could then be used to aggregate fluxes and surface properties from the plants as inputs to models of environmental physics. This would lead to novel hybrid approaches, such as a soil–vegetation–transfer model where the vegetation is replaced by an FSP model.

2.3 Implementation

VPL is implemented as a series of packages written entirely in Julia. In this section, we first justify why Julia is an appropriate choice for an FSP modelling framework. Then, we give an overview of how VPL is implemented in its current iteration, with an emphasis on the public API that users will make use when building models with VPL, as the internal details may change over time.

2.3.1 Why Julia?

By using just-in-time compilation and a sophisticated system for code specialization, Julia achieves high performance while still being dynamic and highly interactive (Bezanson et al. 2017). That is, Julia breaks the 'two language' paradigm that asserts one must choose between slow, dynamic, interactive languages (e.g. Python, R, or Matlab) and fast, compiled, static languages (e.g. C, C++, Fortran, or Java). In simple terms, Julia offers the convenience of Python but also the speed of C++.

By being implemented 100% in Julia and using common data structures, VPL is compatible with existing Julia packages to perform any modelling, simulation, or visualization task beyond those specific to FSP modelling. For example, if the user needs to make use of a particular numerical method (e.g. ODE solver), import or export data, or visualize the data generated by a simulation, they can choose any of the existing Julia packages to perform that task. Furthermore, Julia is highly interoperable with Python and R, so they can also use packages and code written in those languages from within their Julia scripts.

As VPL is written in Julia and supports multiple graphic backends, it is possible to execute VPL code in many different contexts such as (i) remote servers without graphics support; (ii) interactive computational notebooks such as Jupyter (Kluyver et al. 2016), Quarto (Allaire et al. 2022), or Pluto (van der Plas et al. 2023); (iii) IDEs that support Julia such as Visual Studio Code; or (iv) simply by copying code into a terminal. Julia also supports Linux, Windows, MacOS, and FreeBSD operating systems.

Julia and, by extension, VPL are also designed from the beginning for collaborative projects and code reuse. This implies favouring a decentralized, functional approach to programming, as opposed to the more traditional approach of building large software structures with complex class hierarchies. This philosophy has been followed in the development of VPL by (i) reusing existing Julia packages that implement specific functionalities and (ii) allowing users to modify the functionality of VPL as much as possible without touching the source code. Furthermore, we have split the implementation of VPL into multiple packages such that future collaborators can focus on specific aspects of FSP modelling. This approach to software design lowers the entry barrier for advanced users to become co-developers of VPL, favouring the longevity and sustainability of the software.

The internal structure of VPL and the lack of deep hierarchical classes, following Julia's paradigm of multiple dispatch, makes it easier for users to define their own turtle commands, surface materials, and even light sources, as they simply need to inherit from one abstract type and define a few methods per data type.

2.3.2 Overview of VPL and the VPLverse.

VPL is currently implemented as a GitHub organization comprised of:

- Five core packages that implement the basic functionality of the modelling framework (VPLcore)
- An interface package (VirtualPlantLab.jl) that exports the public API users are meant to use
- Four auxiliary packages that further aid in building and running FSP models (VPLverse)
- A collection of tutorials with examples of FSP models illustrating different features
- A website that collects all the documentation regarding VPL

The different repositories and packages are grouped into two sets (Fig. 1). Firstly, the core packages plus interface that form VPL as described in the above. Secondly, auxiliary packages and repositories that complement VPL and meant to be used with it, which we denote as the *VPLverse*.

The five core packages implement the basic data structures and algorithms required to build an FSP model:

- PlantGraphs.jl: A dynamic graph rewriting system where user-defined objects are stored in each node and these

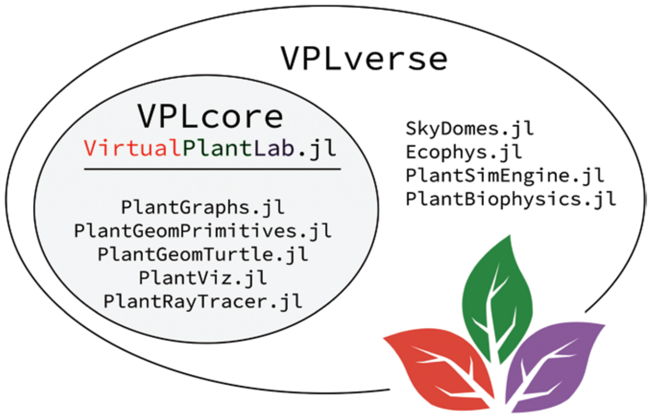

Figure 1. Organizational structure of VPL. The VPLcore includes the interface and five packages that integrate the basic functionality of VPL. Aside from the VPLcore, the VPLverse also includes other packages that are useful for the development of FSP models. All packages are suffixed with '.jl' as is customary in the Julia community.

nodes can be queried and replaced by sub-graphs through dynamic production rules (see Section 2.1.1).

- PlantGeomPrimitives.jl: A collection of 3D primitives implemented as triangular meshes stored in a scene for the purpose of visualization and ray tracing.
- PlantGeomTurtle.jl: An implementation of turtle algorithms that can generate 3D meshes from graphs representing the topology and structure of individual plants.
- PlantRayTracer.jl: A physics-based forward Monte Carlo ray tracer for the purpose of computing light interception by individual plants under field or controlled growth conditions. The ray tracer is multithreaded and uses bounding volume hierarchies.
- PlantViz.jl: 3D rendering of scenes based on the different backends of Makie.jl

The VPLverse also comprises the following packages:

- SkyDomes.jl: A package that generates light sources approximating the distribution of solar radiation in the sky (according to the CIE standards) as well as other variables related to solar geometry, daylength, etc.
- Ecophys.jl: Data structures and functions to simulate different ecophysiological processes, with an emphasis on short-term and long-term responses to aboveground environment (e.g. photosynthesis, transpiration, energy balance, phenology).
- PlantSimEngine.jl: Simulation engine for dynamic, process-based models (Vezy 2023b). It handles model coupling, time stepping, and propagation of physical units or measurement errors through simulations. The package focuses on multiscale simulation and modelling of plants, soil, and atmosphere in interaction. It provides a user-friendly framework for defining processes and implementing associated models. With an emphasis on ease of use and high-performance computations, PlantSimEngine facilitates model coupling and modelling across different scales, from leaf-level simulations to entire scenes. Additionally, it supports various computation methods, including sequential, parallel, and distributed approaches.
- PlantBiophysics.jl: A package to simulate biophysical processes for plants such as photosynthesis, conductance for heat, water vapour and CO_2, latent, sensible energy fluxes, net radiation, and temperature. It leverages PlantSimEngine to declare the processes and implement the models, allowing for fast computations, i.e. just a few microseconds for the whole energy balance, photosynthesis, and conductance coupling. This simulation speed makes it suitable for both model calibration and simulation tasks.

3. EXAMPLE: INTERCROPPING OF CEREAL AND LEGUMES

FSP models are a valuable tool to dissect and analyse the intricate interactions between plants of different and the same crop species (Evers et al. 2019; Gaudio et al. 2019). Therefore, to illustrate the possibilities of VPL, we implemented a model of an intercrop system composed of two species. The simulated plants represent typical cereal and legume crop species, though no formal calibration or parameterization was done. While this model provides a foundation for further development and research to address agroecological or plant physiological questions, exploring these possibilities is beyond the scope of the current paper. Instead, the focus is on providing a comprehensive model description and 3D visualizations of its structure, rather than addressing a particular research question or making predictions about a real crop system. The current model is inspired by and builds upon the model presented by Evers and Bastiaans (2016) . The full source code can be found at https://git.wur.nl/david.kottelenberg/fspm_vpl_dk/-/tree/BASIC_CEREAL_LEGUME and it consists of several files in the Julia programming language (implementing the model) and JSON files (representing the parameters for the different crop species).

Intercropping, the simultaneous cultivation of different crop species on the same land, has potential benefits, including increased yields (Li et al. 2020b) and resource-use efficiency (Chen et al. 2018; Franco et al. 2018), and reduce weed biomass (Gu et al. 2021) when compared to sole crops. A commonly employed intercropping system is cereal–legume row replacement intercropping, featuring alternating rows of cereals and legumes (Naudin et al. 2009; Bedoussac et al. 2014; Kermah et al. 2017) (Supplemenry Fig. S1.1). Intercropping is an example of a heterogeneous canopy, which promotes canopy niche differentiation, potentially resulting in enhanced light capture (Brooker et al. 2015; Li et al. 2020a; Homulle et al. 2021). In response to these environmental cues, plants adapt their growth and developmental strategies to anticipate competition for resources. For instance, cereal plants adapt their elongation rate and tiller production in response to the red-to-far-red light ratio (R:FR ratio), mitigating shading effects (Ballaré and Casal 2000; Evers et al. 2006; Franklin 2008). Furthermore, the competitive balance between the component crop species plays a crucial role in intercrop dynamics, as a dominant species may suppress its companion crop (Hauggaard-Nielsen et al. 2008). Such complex interactions make intercropping an interesting case study for an FSP model.

3.1 Model description

This FSP model simulates, at a daily time step, the growth, development, and architecture of a cereal and a legume grown in the same field. The model describes each individual plant separately and plant-to-plant interactions occur via shading and changes to the light quality.

At the start of a simulation, a `FieldBase` object is created, to store field-level variables at every time step (Fig. 2), which is part of the model output. Each plant is represented by a `Graph` object and the nodes represent either geometric transformations required by the turtle graphics approach or organs (with associated geometry and internal variables). In addition, plant-level data are stored within each graph inside a `PlantBase` object. Information stored in `PlantBase` is used for plant growth, development, and visualization, as well as model output. The simulation output is comprised of the `FieldBase` object and an array of `Graph` objects (with their respective `PlantBase` objects), which are saved for every simulated day. This allows retrieving all the information generated during a simulation at any point, as well as restart any simulation at any point.

Six types of organs are defined: meristem, internode, leaf, root, flower, and fruit. Organs are characterized by different variables (organized in different objects) depending on their function (Fig. 2). All organs contain general organ information in an `OrganInfo` object, organs that can grow (i.e. all organs except the meristem) contain growth parameters in a `GrowthVars` object, and organs that can photosynthesize (i.e. internode and leaf) contain photosynthesis parameters in a `PhotosynthesisVars` object.

In the following sections, we highlight the main aspects of the model regarding development and growth with an emphasis on how we can make use of different features in VPL. More details regarding the model can be found in Supplementary Data 2.

3.1.1 Development.

A new plant starts with a meristem representing the seed. This meristem will produce phytomers over time which represent the basic unit of structure of the aboveground part of the plant. A phytomer comprises of an internode, a leaf with an insertion angle of 50°, and a dormant lateral meristem or bud with an insertion angle of 40°. Phytomers are arranged in a spiral pattern with a rotation angle of 137.5°. Leaf appearance (phyllochron) in cereals is delayed in relation to the leaf initiation rate at the apex,

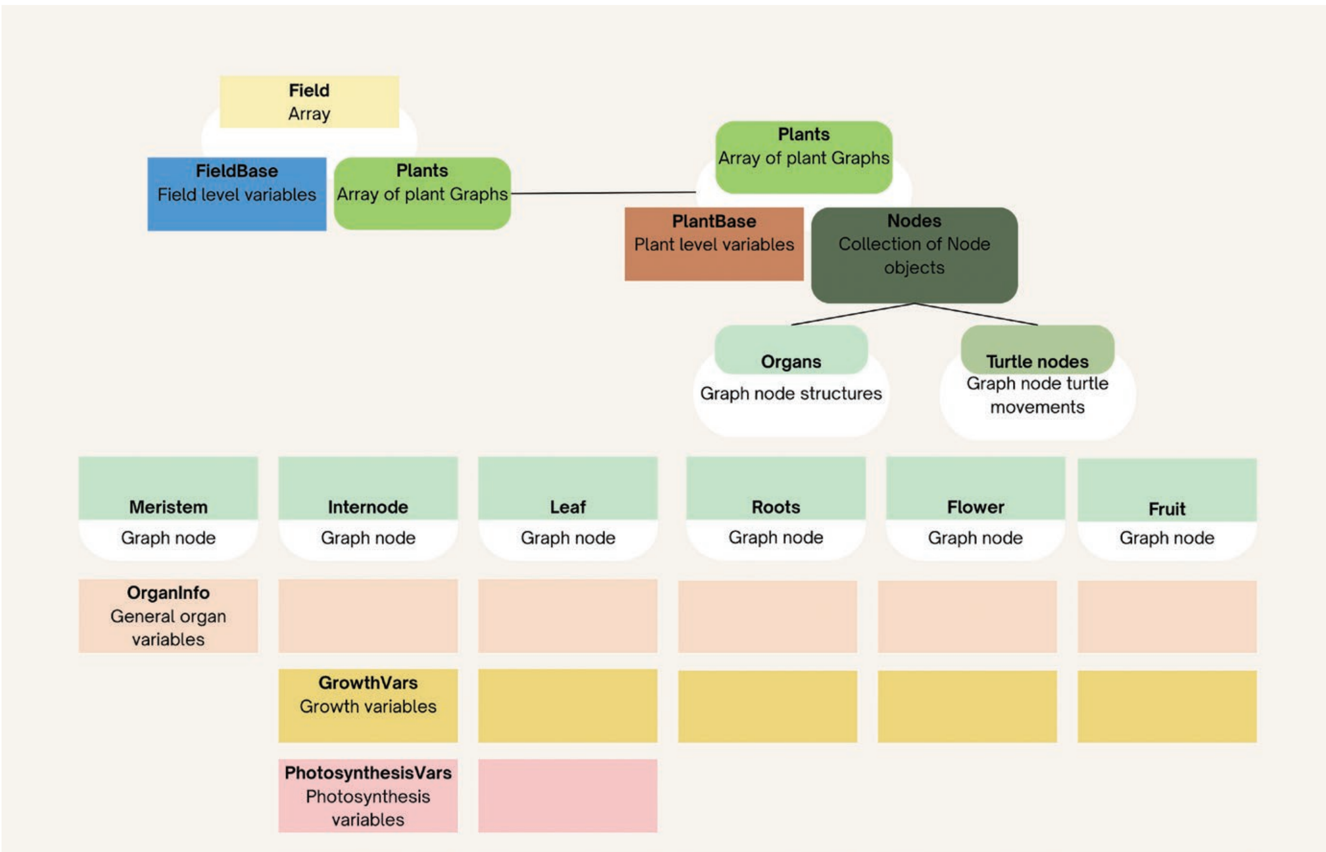

Figure 2. Diagram describing model structures and organization. The simulated field contains a FieldBase object and an array of plants. Each plant is represented by a Graph object containing a PlantBase object and nodes. There are two types of nodes: Turtle movement nodes and Organs. Organs can be one of the following node objects: Meristem, Internode, Leaf, Roots, Flower, or Fruit. The model defines various objects for organ variable storage. Organs may contain one or multiple objects, depending on their functionalities. The specified objects include: OrganInfo (storing general organ variables), GrowthVars (storing growth variables), and PhotosynthesisVars (storing photosynthesis variables).

and set to 90 °Cd per phytomer rank, during which they only act as a sink without active growth. Legume phyllochron is the same as the plastochron, i.e. no delay (see below).

Development of new organs is described by a graph rewriting rule that transforms meristems into new phytomers followed by this same meristem at fixed thermal time intervals (i.e. a plastochron of 45 °Cd and 50°Cd for cereal and legume, respectively). The rule is as follows (we have removed the parameters of each object for clarity):

$$M \rightarrow I + (L, B) + M, \qquad (1)$$

where M is a meristem, I is an internode, and L and B represent a leaf and bud, respectively. In the model, this rule is implemented as

`produce_phytomer_rule = Rule(Meristem,` rhs = produce_phytomer),

where `produce_phytomer` is a function that takes as input each `Meristem` object that will be replaced and returns a replacement graph. In this case, it will produce a graph representing a phytomer. Such graph starts with an `Internode` object and has three children of type `Leaf`, `Bud`, and `Meristem` but only when the plastochron requirement is met. This rule could also have been implemented as:

produce_phytomer_rule = Rule(Meristem, lhs = select_phytomer, rhs = new_phytomer),

where `select_phytomer` would take the `Meristem` object as input and check whether the plastochron requirement is met or not (return a Boolean, i.e. true or false), and then `new_phytomer` would always produce the phytomer. In this variant, the `new_phytomer` is only invoked when `select_phytomer` returns true. This approach would be in better agreement with canonical L-systems but both variants are valid in VPL and lead to the same results, highlighting the flexibility on graph rewriting in VPL.

Another (potential) advantage of the graph rewriting system in VPL is that the left-hand and right-hand side of a production rule can be implemented as functions in a procedural manner. We believe this may make it easier to implement relatively complex rules without having to learn a new mathematical syntax. For example, the `produce_phytomer` function mentioned above is implemented as 130 lines of code as we wanted to account for the different stages of development and the different species in the model within a single rule (see source file `Rules.jl` in the Git repository). Implementing production rules as functions may also help with debugging of models and reuse of code.

3.1.2 Photosynthesis.

The growth dynamics in this model are based on the supply (source) of and demand (sink) for assimilates, after deduction of plant maintenance respiration. Full details on these processes are provided in Supplementary Data 2, but here we emphasize how the supply of assimilates is computed from light absorption and photosynthesis.

Plants gain assimilates through photosynthesis driven by radiation absorption and other environmental conditions. Because light varies during the day and photosynthesis responds to this in a non-linear fashion, a proper integration of the diurnal variation in photosynthesis is required. We chose to perform this integration using a 10-point Gaussian rule (different options are available in Julia, including adaptive integration methods). This means that light absorption is computed at 10 different time points throughout the daytime.

Light interception is also important for morphogenetic processes such as tillering or shade avoidance responses which respond to light quality in addition to quantity. For that reason, the ray tracer allows to simulate different wavebands simultaneously. In the model, we choose to simulate photosynthetic active radiation (PAR) (400–700 nm), red (600–700 nm), and far-red (701–750 nm), all of them expressed in μmol m^{-2} s^{-1}. PAR is used for computing photosynthesis, whereas the other two wavebands allow us calculating the red:far-red ratio use to drive some of the morphological responses (see Supplementary Data 2 for details). Each type of the organ in the crops (as well as the soil) is assigned optical properties with the transmittance and absorptance at each of these wavebands (Supplementary Table S2.1). In all cases, we assumed Lambertian surfaces (i.e. perfect diffusers). This is a limitation as the optical properties of leaves are more complex as light may partially be reflected specularly off leaves. VPL offers the framework to extend this model and includes a Phong model as alternative (Lafortune and Willems 1994), but users can also implement their own bidirectional reflectance/transmittance functions if they wish to do so. These optical properties are bundled in `Material` objects, which will also keep track of the light absorbed by the surface it is assigned to.

Light absorption also varies within the canopy due to self-shading and scattering and, for the same reason as for temporal variation, a proper spatial integration is required. This is achieved in the model by (i) dividing each sub-organ into multiple triangles and (ii) by assigning a separate `Material` object to each triangle. This allows keeping track of the light absorbed by each triangle separately which is then used as input for calculating photosynthesis which is then aggregate to the organ, plant and canopy level. The photosynthesis model for C3 species by Yin and Struik (2009), was used, as implemented in the Ecophys.jl package.

The final component to properly calculate light absorption in the field is to simulate the distribution of light in the sky. A ray tracer requires discrete light source, which is in this case would be implemented as parallel rays of light with specific azimuth and zenith angles (i.e. `Directional` light sources in VPL). Other light sources are offered in VPL to emulate artificial lights (e.g. in greenhouses or vertical farming) but that was not needed in this example.

The more complex step is to ensure that the distribution of light sources approximates the natural distribution of light. To facilitate this, we created the SkyDomes.jl package that will implement several models of angular light distribution in the sky. We distinguish between diffuse and direct light. The former is implemented as a hemispherical dome of light sources (parameterized by the number of subdivisions along the zenith and azimuth angle), whereas direct light is implemented as a single light source. The angles for the direct light source are computed from classic equations of solar trajectory that are common place in crop and FSP models. For the diffuse light, we implement all the CIE standards as well as classic standard clear and overcast skies (Moon and Spencer 1942).

For example, a dome of diffuse light with intensity of 1 with 108 light sources (12 zenith divisions and 9 azimuth divisions) using a standard overcast sky can be generated as follows:

```
dome_diff = sky(scene,
             Idir = (0.0, 0.0, 0.0),
             Idif = (1.0, 1.0, 1.0),
             nrays_dif = 10_000_000,
             sky_model = StandardSky,
             dome_method = equal_solid_
             angles,
             ntheta = 9,
             nphi = 12)
```

where `scene` is a `Scene` object created from all plant graphs, `Idir` and `Idif` are arrays representing the direct and diffuse radiation intensities, respectively, at the three different wavebands mentioned above, `nrays_dif` indicates that 20 million rays will be used to compute diffuse light absorption (to be distributed across all light sources), and the rest of arguments define the sky dome. Note that by specifying `Idir = (0.0, 0.0, 0.0)`, we ensure that we only generate the diffuse light and not the direct light.

In VPL, it is possible to specify multiple sky domes and run the ray tracer for each dome separately. We find this useful as the spatial distribution of diffuse light in typical FSP models does not change during the day so that only has to be computed once (for a value of 1 which than later be scaled according to the diurnal curve of diffuse light). This then allows to only run the ray tracer for direct light at different times of the day, using a dome such as:

```
dome_direct = sky(scene,
             Idir = (direct_PAR, direct_
             red, direct_farred),
             nrays_dir = 100_000,
             Idif = (0.0, 0.0, 0.0),
             theta_dir = theta,
             phi_dir = phi)
```

where `theta` and `phi` (zenith and azimuth angles) and the intensities of direct PAR, red light, and far-red light are calculated from day of year, latitude, and time of day using auxiliary functions from SkyDomes.jl. Being able to separate the ray tracing for diffuse light (once per day) and direct light (several times per day) allows reducing the computational burden of running a ray tracer by avoiding redundant calculations. Please see the source file `Photosynthesis.jl` for more details.

It is also possible to customize the settings of the ray tracer itself via the function `RTSettings` and `accelerate`, such as whether to use the grid cloner or not (and how many clones to generate), whether to run the ray tracer in parallel across multiple cores, the type of acceleration structure, or the parameters of the Russian roulette. For example, in our model we used:

```
settings = RTSettings(pkill = 0.9,
                      maxiter = 5,
                      nx = 5,
                      ny = 5,
                      parallel = true)
acceleration_struct = accelerate(scene,
                      settings      =
                      settings,
                      acceleration  =
                      BVH,
                      rule = SAH{6}(5,
                      15))
```

where `pkill` is the probability that a ray is terminated by the Russian roulette after it has been scattered a `maxiter` number of times, `nx` and `ny` indicate how many times the grid is cloned in the x and y direction (to avoid edge effects), `parallel` indicates whether multiple threads can be used in parallel to speed up the ray tracing, `BVH` standing for Bounding Volume Hierarchy as the acceleration structure, and `SAH` stands for Surface Area Heuristics as the rule to split triangles into different bounding volumes (see VPL documentation for more details).

Everything is bundled into a ray tracing object and the actual ray tracing is performed with the function `trace!`:

```
ray_tracer_object                          =
RayTracer(acceleration_struct,   mate-
rials(scene), dome, settings)
  trace!(ray_tracer_object)
```

In our simulation, we run the ray tracer 11 times per day, one using the dome for diffuse light (with 10 million rays) and 10 times using a direct light dome for a different time of the day (with a hundred thousand rays each) for a total of 20 million rays per simulated day. From this, we compute the photosynthesis of each triangle, sub-organ, plant or canopy at each time of the day.

3.2 Visualization framework and application

3.2.1 Geometry construction and the feed! function.

When a scene is constructed using one or more `Graph` objects representing plants, the `feed!` function is called on the graph. Each type of object used in the graph (i.e. either operators for turtle movements or user-defined objects such as the ones in Fig. 2) defines its own method for the `feed!` function. The result of calling `feed!` on the graph is that it will visit every node in the graph and call the corresponding `feed!` method on each node (depending on the type of data stored in it). The different `feed!` methods are then responsible for generating 3D meshes corresponding to the different organs and feeding them to the turtle (hence the name) that will keep track of all these meshes. These methods can also alter the position and orientation of the turtle as it traverses the graph.

For example, the `feed!` method for an `Internode` object is implemented as follows:

```
function    PlantGeomTurtle.feed!(tur-
tle::Turtle, i::Internode, data)
   sp = data.species_parameters
    color  =  get_color(i,  turtle.mes-
sage, data)
```

```
HollowCylinder!(turtle,
                length = i.length,
                height = i.width/2,
                width = i.width/2,
                move = true,
                colors = RGBA(color[1], color[2], color[3], color[4]),
                materials = i.material)
    return nothing
end
```

where the `Turtle` object contains the position and orientation of the turtle. The internode is constructed using the `HollowCylinder!` function, which defines its dimensions (length, height, width), colour (for visualization), and

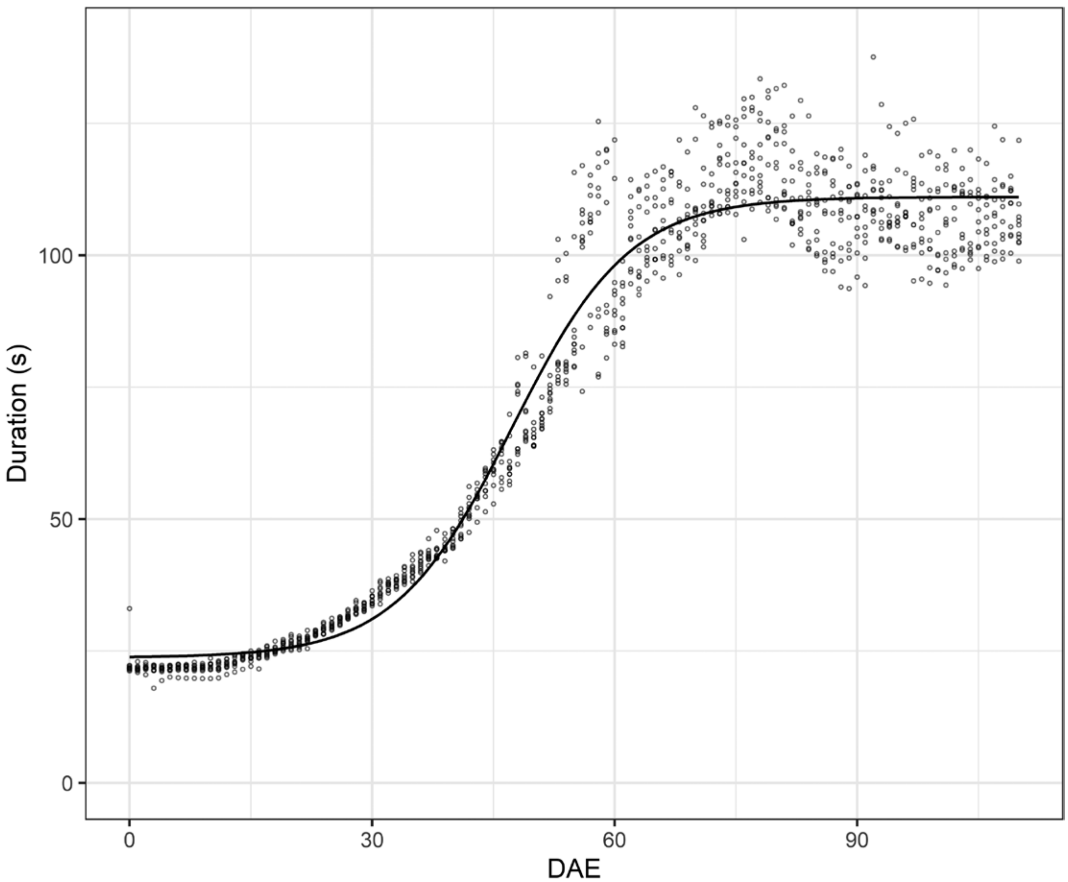

Figure 3. Duration of a simulated day in the model (in seconds) plotted against simulation DAE across 10 simulations, with a logistic curve fitted to the data.

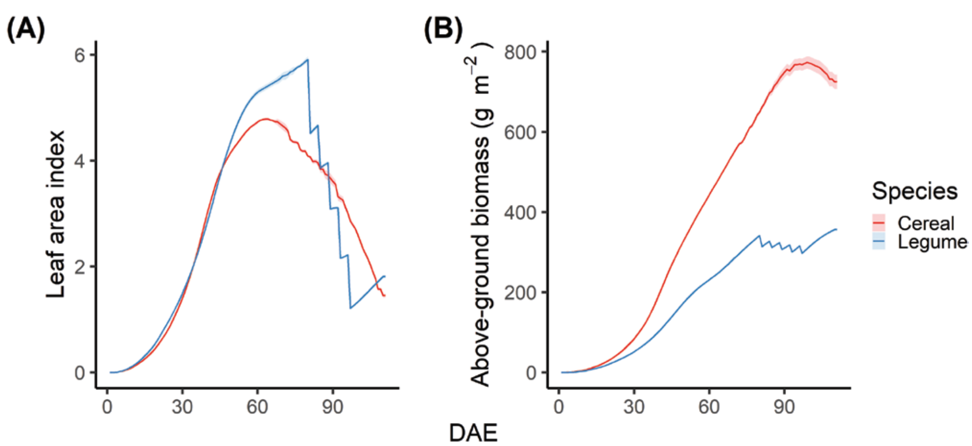

Figure 4. Leaf area index (A) and aboveground biomass (B, g m^{-2}) for cereal and legume in the simulated intercrop. The shaded area indicates the standard deviation.

optical properties (`materials`, for ray tracing) stored in the Internode object. The `move` argument indicates whether the turtle should move to the end of the newly created geometry after constructing the mesh or remain at its current position. The `get_color` function is an user-defined function that chooses the colour to be assigned to the organ based on the `Turtle` message (see Supplementary Data 2). This allows different visualizations of the same scene and to link the 3D visualizations of the same scene (see below for details).

3.2.2 Performance benchmarking.

To test the current model, we benchmarked a cereal–legume intercrop with alternating rows of cereal and legume plants. Densities were set at 240 cereal plants m^{-2} and 64 legume plants m^{-2} field area. Row distances were set at 0.125 m and plot size remained constant at 2.0 × 0.5 m in all simulations. To mitigate border effects, plots were cloned five times in every cardinal direction (with the VPL built-in grid cloner, see code snippet on `RTSettings` above), resulting in a field with 11 × 11 clones of the original plot. The ray tracer was setup with 20 million total

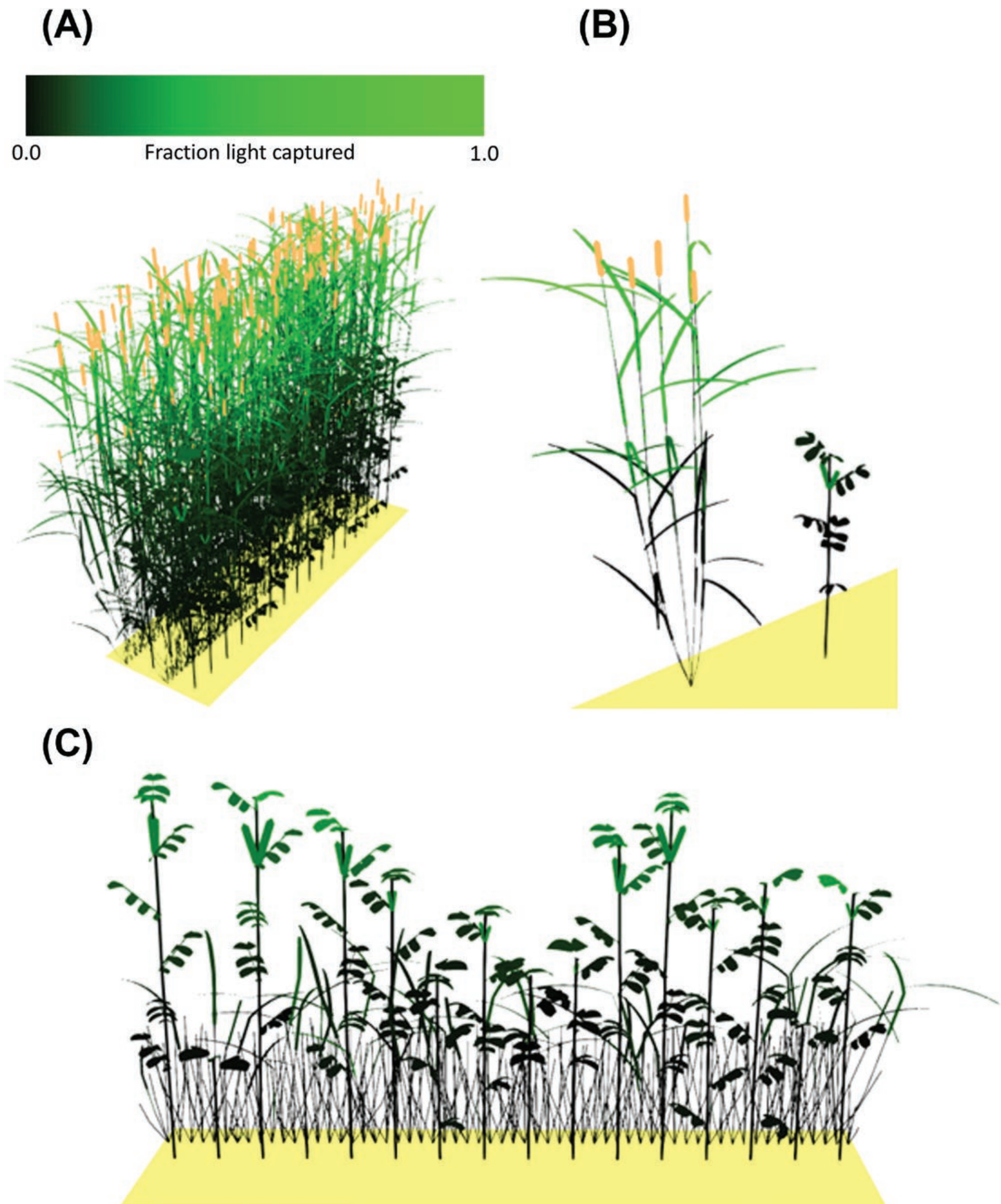

Figure 5. Render outputs with colours based on fraction of light captured. (A) Field render output of the normal density cereal–legume intercrop treatment at 100 DAE. (B) Rendering of a single cereal and legume plant from the field in (A). (C) Partial view of the field in (A), featuring only two rows, with cereal organs having a cumulative phytomer number exceeding six rendered transparent to enhance visibility of the upper sections of legume plants.

rays which in our experience is more than sufficient to ensure reasonable integration errors in the calculation of crop growth.

We tested our model on a system equipped with an AMD Ryzen Threadripper 7980X processor, utilizing all 64 physical cores (128 threads) for 64 GB of available RAM. Based on an average of ten runs, simulation runtimes ranged between 1:54:30 and 2:05:06 (h:m:s). The runtime per simulated day increased with time after emergence, following a logistic pattern, plateauing once phytomer creation ceased for cereals and leaves began senescing (Fig. 3). Runtime variance increases as a simulation progresses, likely driven by small stochastic differences early in the simulation that amplify over time. Using a profiler, we measured that the ray tracer accounted for over 99.9% of the total runtime, clearly identifying it as the primary bottleneck in the simulation process. This may explain the logistic pattern we saw in runtime, as the time it takes to construct the 3D meshes and ray trace them scales with the number of foliar elements and therefore simulated leaf area. These results indicate that improving the performance of the ray tracer is likely to greatly improve the runtime of FSP models built with VPL. Further development of VPL will explore the possibility of accelerating the ray tracer with the use of GPUs and optimizing the number of rays needed for different levels of accuracy in simulations.

Although our model is not calibrated to field data, the species parameters are roughly based on triticale and faba bean. Our model simulates crop behaviour with outputs that realistically reflect the growth characteristics of these species. The simulated legume leaf area index (LAI; leaf area per unit field area) approaches 6.0 before declining (Fig. 4A), similar to values

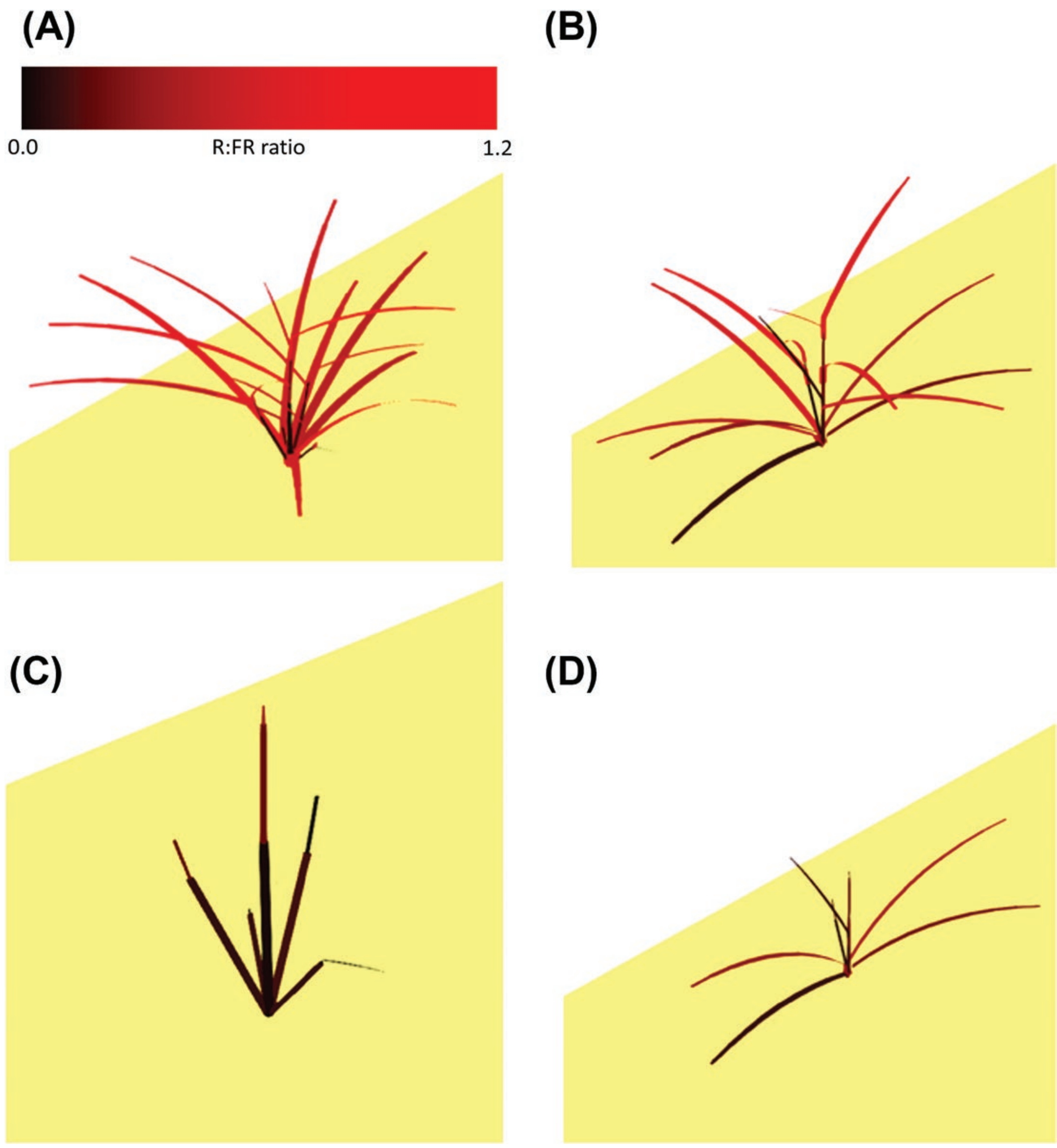

Figure 6. Individual cereal plants rendered from a fully simulated field at 40 DAE, based on organ red:far-red ratio (neighbouring plants removed for visualization purposes). (A) A cereal plant from the cereal–legume treatment. (B) A cereal plant from the high-density cereal sole crop treatment. (C and D) The same plants as in (A) and (B) respectively, but with organs having an R:FR ratio exceeding 0.4 removed for visualization purposes.

found for faba bean in sole crops at 35 plants m^{-2} (Ebrahimi et al. 2024). This is roughly half of our legume density of 64 plants m^2, making the comparison valid despite the different planting systems. Cereal LAI approaches 4.5 (Fig. 4A), comparable to triticale in sole crops at about 62% of our density, suggesting a realistic comparison between our intercrop and monocrop systems (Nielsen et al. 2012). Aboveground biomass reaches 800 g m^{-2} for the cereal and 300 g m^{-2} for the legume (Fig. 4B), which aligns with values reported for various cereal–faba bean intercrops (Lithourgidis and Dordas 2010). The zig–zag pattern in the legume lines in both figures results from leaf senescence, which lacks significant stochasticity in this model, causing relatively drastic decreases in the variables.

3.2.3 *Visualization flexibility.*

The `render` function creates a visualization of the whole scene using the colours assigned within the `feed!` methods. VPL allows for different visualization of the same graphs by using a `message` passed to these `feed!` methods. In our model, we linked the colours to specific plant characteristics using an ad-hoc function `get_color` (see Supplementary Data 2):

- Radiation capture: Darker greens indicate reduced radiation interception.
- R:FR ratio: Redder organs represent lower red:far-red light ratios.
- Selective visualization: Filter organs above or below a given phytomer rank or R:FR threshold.

These visualization features enable intuitive exploration of plant competitiveness in intercrops. At 100 days after emergence (DAE), the canopy exhibits a distinct gradient of radiation capture along its height (Fig. 5A). Cereal plants appear brighter than legume plants, reflecting their superior radiation capture, particularly at the top of the canopy (Fig. 5B). To enhance visibility of legume plants' upper sections, the visualization can filter out cereal phytomers with a rank exceeding six from the plant base, focusing on two rows of the rendered field:

```
scene = Scene(field[2], message =
["light_capture", "remove_upper_
phyt", "6", "cereal"])
  display(render(scene))
```

where `field[2]` contains the plant `Graph` objects, and the `message` (to be processed by the function `get_color`) indicates green colouring based on light capture, with cereal phytomers above rank 6 removed (Fig. 5C). This highlights how taller legume plants capture more light, shown in their brighter tops compared to shorter legumes. Such visualizations showcase VPL's capability for nuanced 3D rendering.

Additionally, R:FR ratio-based visualizations offer insights into early plant responses to competition. Unlike light capture, which directly indicates absorbed light, R:FR ratio serves as an early indicator of potential shading. Plants detect reduced R:FR light reflected from neighbours before shading occurs. For example, visualizations comparing R:FR ratios at 40 DAE under conditions of low and high competition illustrate these differences. Under lower competition, cereal plants exhibit lighter-coloured organs (Fig. 6A), whereas under higher competition, darker-coloured organs predominate (Fig. 6B). By further restricting visualization to organs with an R:FR ratio below 0.4, differences become even more pronounced, e.g.:

```
scene = Scene(field[2], message =
["rfr", "remove_rfr_above", "0.4",
"cereal"])
  display(render(scene))
```

From this visualization, we see that in low-competition conditions, no leaves fall into this category (Fig. 6C), whereas high-competition conditions show several such leaves (Fig. 6D).

3.3 Concluding remarks

With this example, we demonstrated that VPL is suitable to simulate intercrops using FSP modelling, and it can capture many of the plant–environment interactions including photosynthesis, red:far-red ratio, or competition among plants for light. The advanced visualization system in VPL is particularly useful for gaining insights into the simulations, whereas the inclusion of a ray tracer allows for general purpose simulation of light interception without restrictive assumptions about the system. We are currently extending the intercrop model to account for roots and microclimate and further developing VPL in parallel to ensure that such type of modelling is also well supported.

SUPPLEMENTARY DATA

Supplementary data are available at *in silico Plants* online.

AUTHOR CONTRIBUTIONS

A. Morales has written all the source code behind the core of VPL and most packages in the VPLverse (excluding the ones by R. Vezy). R. Vezy wrote the source code behind the packages PlantSimEngine.jl and PlantBiophysics.jl and provided valuable feedback on the core of VPL. A. Ernst has contributed to the documentation of VPL, organized the website, designed the official logo, and provided support to A. Morales during the development of VPL. D. B. Kottelenberg wrote the model used as example in this paper and ran the simulations with supervision by J. B. Evers. All authors contributed to writing the paper.

CONFLICT OF INTEREST

None declared.

FUNDING

D.B.K. was funded by the Synergia project. The Synergia project is organized and led by Wageningen University and Research in close cooperation with Next Food Collective as well as the Universities of Delft, Twente, Eindhoven, and Nijmegen. Funding for this research was obtained from the Netherlands Organisation for Scientific Research (NWO grant 17626), IMEC-One Planet, and other private parties.

DATA AVAILABILITY

All code is freely available on a standard MIT license (no need to contact the authors). All source code for the Virtual Plant Laboratory is hosted in the following GitHub organization: https://github.com/VirtualPlantLab. The source code for the model presented as example can be found here: https://git.wur.nl/david.kottelenberg/fspm_vpl_dk/-/tree/BASIC_CEREAL_LEGUME. For instructions on installation, we refer to the official website (which contains additional examples with their source code): https://virtualplantlab.com.

REFERENCES

Allaire JJ, Teague C, Scheidegger C, et al. 2022. *Quarto*. 1.2 ed. https://doi.org/10.5281/zenodo.5960048.

Bailey BN. A reverse ray-tracing method for modelling the net radiative flux in leaf-resolving plant canopy simulations. *Ecol Model* 2018;368:233–45. https://doi.org/10.1016/j.ecolmodel.2017.11.022

Baldocchi DD, Wilson KB, Gu L. How the environment, canopy structure and canopy physiological functioning influence carbon, water and energy fluxes of a temperate broad-leaved deciduous forest—an assessment with the biophysical model CANOAK. *Tree Physiol* 2002;22:1065–77. https://doi.org/10.1093/treephys/22.15-16.1065

Ballaré CL, Casal JJ. Light signals perceived by crop and weed plants. *Field Crops Res* 2000;67:149–60. https://doi.org/10.1016/s0378-4290(00)00090-3

Bedoussac L, Journet E, Hauggaard-Nielsen H, et al. Eco-functional intensification by cereal-grain legume intercropping in organic farming systems for increased yields, reduced weeds and improved grain protein concentration. In: Bellon S, Penvern S (eds.), *Organic Farming, Prototype for Sustainable Agricultures*. Dordrecht, the Netherlands: Springer, 2014, 47–63. https://doi.org/10.1007/978-94-007-7927-3

Bezanson J, Edelman A, Karpinski S et al. Julia: a fresh approach to numerical computing. *SIAM Rev* 2017;59:65–98. https://doi.org/10.1137/141000671

Brooker RW, Bennett AE, Cong WF et al. Improving intercropping: a synthesis of research in agronomy, plant physiology and ecology. *New Phytol* 2015;206:107–17. https://doi.org/10.1111/nph.13132

Buck-Sorlin G. Functional-structural plant modeling. In: Dubitzky W, Wolkenhauer O, Cho K-H, et al. (eds), *Encyclopedia of Systems Biology*. New York, NY: Springer New York, 2013. https://doi.org/10.1007/978-1-4419-9863-7_1479.

Buck-Sorlin G, de Visser PHB, Henke M et al. Towards a functional–structural plant model of cut-rose: simulation of light environment, light absorption, photosynthesis and interference with the plant structure. *Ann Bot (Lond)* 2011;108:1121–34. https://doi.org/10.1093/aob/mcr190

Chelle M, Andrieu B. The nested radiosity model for the distribution of light within plant canopies. *Ecol Model* 1998;111:75–91. https://doi.org/10.1016/s0304-3800(98)00100-8

Chen G, Kong X, Gan Y et al. Enhancing the systems productivity and water use efficiency through coordinated soil water sharing and compensation in strip-intercropping. *Sci Rep* 2018;8:10494. https://doi.org/10.1038/s41598-018-28612-6

Chen Q, Baldocchi D, Gong P et al. Modeling radiation and photosynthesis of a heterogeneous savanna woodland landscape with a hierarchy of model complexities. *Agric For Meteorol* 2008;148:1005–20. https://doi.org/10.1016/j.agrformet.2008.01.020

Cieslak M, Lemieux C, Hanan J et al. Quasi-Monte Carlo simulation of the light environment of plants. *Functional Plant Biol* 2008;35:837–49. https://doi.org/10.1071/FP08082

Couvreur V, Faget M, Lobet G et al. Going with the flow: multiscale insights into the composite nature of water transport in roots. *Plant Physiol* 2018;178:1689–703. https://doi.org/10.1104/pp.18.01006

de Reffye P, Hu B, Kang M et al. Two decades of research with the GreenLab model in agronomy. *Ann Bot (Lond)* 2021;127:281–95. https://doi.org/10.1093/aob/mcaa172

Duursma RA, Medlyn BE. MAESPA: a model to study interactions between water limitation, environmental drivers and vegetation function at tree and stand levels, with an example application to $[CO_2] \times$ drought interactions. *Geosci Model Dev* 2012;5:919–40. https://doi.org/10.5194/gmd-5-919-2012

Ebrahimi N, Salihy AR, Alipour S et al. Prediction of distribution of dry matter and leaf area of faba bean (*Vicia faba*) using nonlinear regression models. *Agric Res* 2024;13:381–9. https://doi.org/10.1007/s40003-024-00700-2

Evers, JB, Bastiaans, L. Quantifying the effect of crop spatial arrangement on weed suppression using functional-structural plant modelling. *J Plant Res*, 2016;129:339–351. https://doi.org/10.1007/s10265-016-0807-2

Evers JB, van der Werf W, Stomph TJ et al. Understanding and optimizing species mixtures using functional-structural plant modelling. *J Exp Bot* 2019;70:2381–8. https://doi.org/10.1093/jxb/ery288

Evers JB, Vos J, Andrieu B et al. Cessation of tillering in spring wheat in relation to light interception and red: far-red ratio. *Ann Bot (Lond)* 2006;97:649–58. https://doi.org/10.1093/aob/mcl020

Farquhar GD, von Caemmerer S, Berry JA. A biochemical model of photosynthetic CO_2 assimilation in leaves of C3 species. *Planta* 1980;149:78–90. https://doi.org/10.1007/BF00386231

Franco JG, King SR, Volder A. Component crop physiology and water use efficiency in response to intercropping. *Eur J Agron* 2018;93:27–39. https://doi.org/10.1016/j.eja.2017.11.005

Franklin KA. Shade avoidance. *New Phytol* 2008;179:930–44. https://doi.org/10.1111/j.1469-8137.2008.02507.x

Gaudio N, Escobar-Gutiérrez AJ, Casadebaig P et al. Current knowledge and future research opportunities for modeling annual crop mixtures. A review. *Agron Sustainable Dev* 2019;39:20. https://doi.org/10.1007/s13593-019-0562-6.

Godin C, Caraglio Y. A multiscale model of plant topological structures. *J Theor Biol* 1998;191:1–46. https://doi.org/10.1006/jtbi.1997.0561

Godin C, Sinoquet H. Functional–structural plant modelling. *New Phytol* 2005;166:705–8. https://doi.org/10.1111/j.1469-8137.2005.01445.x

Gu C, Bastiaans L, Anten NPR et al. Annual intercropping suppresses weeds: a meta-analysis. *Agric Ecosyst Environ* 2021;322:107658. https://doi.org/10.1016/j.agee.2021.107658.

Hauggaard-Nielsen H, Jørnsgaard B, Kinane J et al. Grain legume–cereal intercropping: the practical application of diversity, competition and facilitation in arable and organic cropping systems. *Renewable Agric Food Syst* 2008;23:3–12. https://doi.org/10.1017/s1742170507002025

Hemmerling R, Kniemeyer O, Lanwert D et al. The rule-based language XL and the modelling environment GroIMP illustrated with simulated tree competition. *Functional Plant Biol* 2008;35:739–50. https://doi.org/10.1071/FP08052

Ho QT, Berghuijs HNC, Watté R et al. Three-dimensional microscale modelling of CO_2 transport and light propagation in tomato leaves enlightens photosynthesis. *Plant Cell Environ* 2016;39:50–61. https://doi.org/10.1111/pce.12590

Homulle Z, George TS, Karley AJ. Root traits with team benefits: understanding belowground interactions in intercropping systems. *Plant Soil* 2021;471:1–26. https://doi.org/10.1007/s11104-021-05165-8

Kent ER, Bailey BN. The role of spatial averaging scale in leaf-to-canopy scaling of non-linear processes in homogeneous canopies. *J Geophys Res Biogeosci* 2021;126:e2020JG005796. https://doi.org/10.1029/2020JG005796

Kermah M, Franke AC, Adjei-Nsiah S et al. Maize-grain legume intercropping for enhanced resource use efficiency and crop productivity in the Guinea savanna of northern Ghana. *Field Crops Res* 2017;213:38–50. https://doi.org/10.1016/j.fcr.2017.07.008

Kluyver T, Ragan-Kelley B, Pérez F et al. Jupyter Notebooks—a publishing format for reproducible computational workflows. In: Loizides F, Schmidt B (eds), *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, 2016, 87–90.

Kurth W, Kniemeyer O, Buck-Sorlin G. *Relational Growth Grammars—A Graph Rewriting Approach to Dynamical Systems with a Dynamical Structure*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2005.

Lafortune EP, Willems YD. *Using the Modified Phong Reflectance Model for Physically Based Rendering. Technical Report CW 197*. Leuven, Belgium: Departement Computerwetenschappen, Katholieke Universiteit Leuven, 1994.

Li C, Hoffland E, Kuyper TW et al. Yield gain, complementarity and competitive dominance in intercropping in China: a meta-analysis of drivers of yield gain using additive partitioning. *Eur J Agron* 2020a;113:125987. https://doi.org/10.1016/j.eja.2019.125987

Li C, Hoffland E, Kuyper TW et al. Syndromes of production in intercropping impact yield gains. *Nat Plants* 2020b;6:653–60. https://doi.org/10.1038/s41477-020-0680-9

Lindner M, Lincoln L, Drauschke F et al. NetworkDynamics.jl—composing and simulating complex networks in Julia. *Chaos* 2021;31:063133. https://doi.org/10.1063/5.0051387

Lithourgidis AS, Dordas CA. Forage yield, growth rate, and nitrogen uptake of faba bean intercrops with wheat, barley, and rye in three seeding ratios. *Crop Sci* 2010;50:2148–58. https://doi.org/10.2135/cropsci2009.12.0735

Louarn G, Song Y. Two decades of functional-structural plant modelling: now addressing fundamental questions in systems biology and predictive ecology. *Ann Bot (Lond)* 2020;126:501–9. https://doi.org/10.1093/aob/mcaa143

MacDonald JD, Booth KS. Heuristics for ray tracing using space subdivision. *Visual Computer* 1990;6:153–66. https://doi.org/10.1007/bf01911006

Marschner S, Shirley P. *More Ray Tracing. Fundamentals of Computer Graphics*. New York: A K Peters/CRC Press, 2018.

Moon P, Spencer DE. Illumination from a non-uniform sky. *Illumination Eng* 1942;37:707–26.

Naudin C, Aveline A, Corre-Hellou G et al. Agronomic analysis of the performance of spring and winter cereal-legume intercrops in organic agriculture. *J Agric Sci Technol* 2009;3:17–28.

Nielsen DC, Miceli-Garcia JJ, Lyon DJ. Canopy cover and leaf area index relationships for wheat, triticale, and corn. *Agron J* 2012;104:1569–73. https://doi.org/10.2134/agronj2012.0107n.

Pearcy RW, Muraoka H, Valladares F. Crown architecture in sun and shade environments: assessing function and trade-offs with a three-dimensional simulation model. *New Phytol* 2005;166:791–800. https://doi.org/10.1111/j.1469-8137.2005.01328.x

Postma JA, Kuppe C, Owen MR et al. OpenSimRoot: widening the scope and application of root architectural models. *New Phytol* 2017;215:1274–86. https://doi.org/10.1111/nph.14641

Pradal C, Dufour-Kowalski S, Boudon F et al. OpenAlea: a visual programming and component-based software platform for plant modeling. *Functional Plant Biol* 2008;35:751–60. https://doi.org/10.1071/FP08084

Prusinkiewicz P, Karwowski R, Měch R, et al. *L-Studio/cpfg: A Software System for Modeling Plants*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2000.

Prusinkiewicz P, Lindenmayer A. *Graphical Modeling Using L-Systems. The Algorithmic Beauty of Plants*. New York, NY: Springer New York, 1990. https://doi.org/10.1007/978-1-4613-8476-2_1.

Retta MA, Yin X, Ho QT et al. The role of chloroplast movement in C4 photosynthesis: a theoretical analysis using a three-dimensional reaction–diffusion model for maize. *J Exp Bot* 2023;74:4125–42. https://doi.org/10.1093/jxb/erad138

Sinoquet H, Le Roux X, Adam B et al. RATP: a model for simulating the spatial distribution of radiation absorption, transpiration and photosynthesis within canopies: application to an isolated tree crown. *Plant Cell Environ* 2001;24:395–406. https://doi.org/10.1046/j.1365-3040.2001.00694.x

Smith RS, Guyomarc'h S, Mandel T et al. A plausible model of phyllotaxis. *Proc Natl Acad Sci USA* 2006;103:1301–6. https://doi.org/10.1073/pnas.0510457103

van der Plas F, Dral M, Berg P, et al. fonsp/Pluto.jl: v0.19.30. v0.19.30 ed: Zenodo, 2023.

van Ittersum MK, Leffelaar PA, van Keulen H et al. On approaches and applications of the Wageningen crop models. *Eur J Agron* 2003;18:201–34. https://doi.org/10.1016/s1161-0301(02)00106-5

Verhoeff T. 3D turtle geometry: artwork, theory, program equivalence and symmetry. *Int J Arts Technol* 2010;3:288–319. https://doi.org/10.1504/ijart.2010.032569

Vezy R. MultiScaleTreeGraph.jl: read, write, analyze, compute and plot multi-scale tree graph files. v0.12.0 ed: Zenodo, 2023a.

Vezy R. PlantSimEngine: a simulation engine for the soil-plant-atmosphere system. *J Open Source Software* 2023b;8:5371. https://doi.org/10.21105/joss.05371

Vezy R, Perez R, Grand F, et al. Light exchanges in discrete directions as an alternative to raytracing and radiosity. In: FSPM 2020: Towards Computable Plants. 9th International Conference on Functional-Structural Plant models, Hannover, Germany, 2020. https://doi.org/10.13140/RG.2.2.13421.20968/1.

Vos J, Evers JB, Buck-Sorlin GH et al. Functional–structural plant modelling: a new versatile tool in crop science. *J Exp Bot* 2009;61:2101–15. https://doi.org/10.1093/jxb/erp345

Wang E, Robertson MJ, Hammer GL et al. Development of a generic crop model template in the cropping system model APSIM. *Eur J Agron* 2002;18:121–40. https://doi.org/10.1016/s1161-0301(02)00100-4

Wang Y, Song Q, Jaiswal D et al. Development of a three-dimensional ray-tracing model of sugarcane canopy photosynthesis and its application in assessing impacts of varied row spacing. *Bioenergy Res* 2017;10:626–34. https://doi.org/10.1007/s12155-017-9823-x.

Yin X, Struik PC. C3 and C4 photosynthesis models: an overview from the perspective of crop modelling. *NJAS* 2009;57:27–38. https://doi.org/10.1016/j.njas.2009.07.001

Zhang B, DeAngelis DL. An overview of agent-based models in plant biology and ecology. *Ann Bot (Lond)* 2020;126:539–57. https://doi.org/10.1093/aob/mcaa043